

UML 2

UMA ABORDAGEM PRÁTICA

A UML – Unified Modeling Language ou Linguagem de Modelagem Unificada – é uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objetos. Nos últimos anos, a UML consagrou-se como a linguagem-padrão de modelagem adotada pela indústria de Engenharia de Software, havendo atualmente um amplo mercado para profissionais que a dominem.

Este livro procura ensinar ao leitor, por meio de exemplos práticos, como modelar softwares com a UML. A linguagem é ensinada mediante a apresentação de seus diversos diagramas, onde são detalhados os componentes de cada diagrama e como estes interagem. Também é demonstrado, por meio de diversas ilustrações, como utilizar cada diagrama. Além disso, o livro mostra como mapear classes em tabelas de banco de dados relacionais, enfocando a questão de persistência. A obra enfatiza ainda a importância da UML para a Engenharia de Software, além de abordar o paradigma de orientação a objetos, um conceito imprescindível para a compreensão correta da UML.

O livro apresenta vários exercícios, como forma de avaliar e consolidar os conhecimentos adquiridos pelo leitor, com as respectivas soluções ao final do capítulo onde foram propostos. Há ainda um estudo de caso, no qual um sistema é analisado e modelado por meio da UML, com a ilustração completa de todos os diagramas referentes ao software.

A obra pode ser utilizada tanto por professores e alunos universitários, de cursos da área de computação, quanto por profissionais da área de Engenharia e desenvolvimento de software.

sobre o autor

Gilleanes Thorwald Araujo Guedes é mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS) e bacharel em Informática pela Universidade da Região da Campanha (URCAMP). É professor de Engenharia de Software no curso de Licenciatura Plena em Informática na Universidade Federal de Mato Grosso (UFMT), estando atualmente cursando o doutorado em Ciência da Computação na Universidade Federal do Rio Grande do Sul. Já ministrou diversas palestras e cursos sobre UML em eventos científicos, cursos técnicos e cursos de pós-graduação "latu sensu". É autor dos livros *UML – Uma abordagem prática*, *UML 2 – Guia de consulta rápida* e *UML 2 – Guia prático*, publicados pela Novatec Editora. Pode ser contatado pelo e-mail gtag@novatec.com.br.

ISBN 978-85-7522-281-2



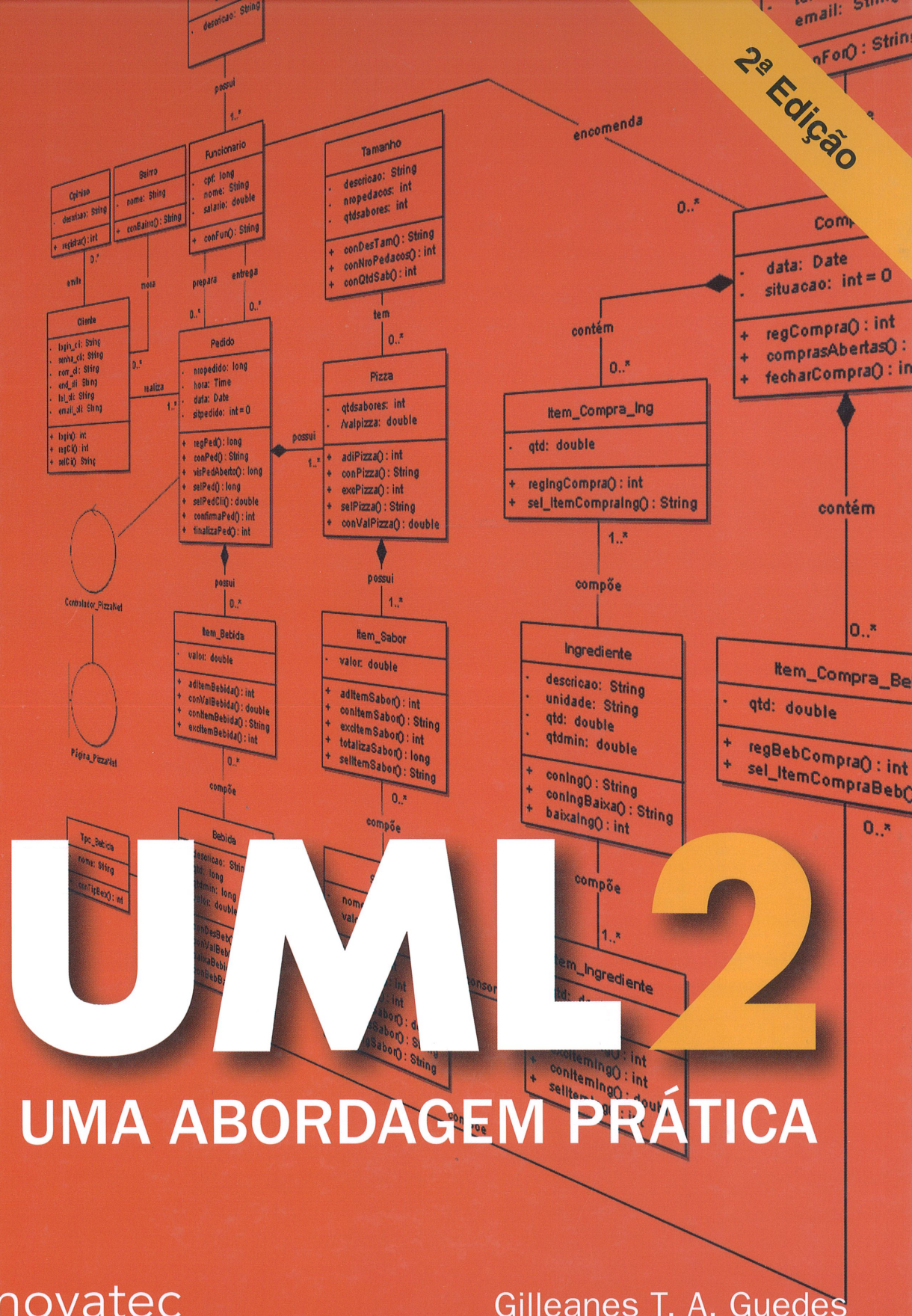
novatec editora

Gilleanes T. A. Guedes

UML 2

UMA ABORDAGEM PRÁTICA

novatec



novatec

Gilleanes T. A. Guedes

VISIT...

LANZAROTE
Caliente.COM

UML 2

UMA ABORDAGEM PRÁTICA

2ª EDIÇÃO

UML 2

UMA ABORDAGEM PRÁTICA

Gilleanes T. A. Guedes

Novatec

Copyright © 2009, 2011 da Novatec Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Revisão de texto: Lia Gabriele Regius

Editoração eletrônica: Camila Kuwabata e Carolina Kuwabata

Capa: Victor Bittow

ISBN: 978-85-7522-281-2

Esta obra foi revisada, atualizada e ampliada tomando como base o livro:

“UML – Uma Abordagem Prática”, ISBN 978-7522-149-5

Histórico de impressões:

Junho/2011

Segunda edição

Maior/2009

Primeira edição (ISBN: 978-85-7522-193-8)

NOVATEC EDITORA LTDA.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Fax: +55 11 2950-8869

E-mail: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

*Dedico este livro à minha esposa, Sílvia,
e aos meus filhos, Ulisses e Sophia.*

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Guedes, Gilleanes T. A.
UML 2 : uma abordagem prática / Gilleanes T. A.
Guedes. -- 2. ed. -- São Paulo :
Novatec Editora, 2011.

Bibliografia.
ISBN 978-85-7522-281-2

1. Componentes - Programas de computador
2. Programação orientada para o objeto (Ciência da
computação) 3. Software - Desenvolvimento 4. UML
(Ciência da computação) I. Título.

07-9482

CDD-005.3

Índices para catálogo sistemático:

1. Linguagem de modelagem unificada :
Ciência da computação 005.3
2. UML : Unified Modeling Language : Programas :
Ciência da computação 005.3
CRS20110602

Sumário

Agradecimentos	15
Sobre o Autor	16
Prefácio	17
Capítulo 1 ■ Introdução à UML	19
1.1 Breve Histórico da UML.....	19
1.2 Por Que Modelar Software?	20
1.2.1 Modelo de Software – Uma Definição.....	21
1.2.2 Levantamento e Análise de Requisitos	21
1.2.3 Prototipação	24
1.2.4 Prazos e Custos	25
1.2.5 Projeto	26
1.2.6 Manutenção	27
1.2.7 Documentação Histórica	29
1.3 Por que tantos Diagramas?.....	30
1.4 Rápido Resumo dos Diagramas da UML	30
1.4.1 Diagrama de Casos de Uso	30
1.4.2 Diagrama de Classes.....	31
1.4.3 Diagrama de Objetos	32
1.4.4 Diagrama de Pacotes.....	33
1.4.5 Diagrama de Sequência.....	33
1.4.6 Diagrama de Comunicação	35
1.4.7 Diagrama de Máquina de Estados	35
1.4.8 Diagrama de Atividade.....	36
1.4.9 Diagrama de Visão Geral de Interação	37
1.4.10 Diagrama de Componentes	38
1.4.11 Diagrama de Implantação	39
1.4.12 Diagrama de Estrutura Composta	40
1.4.13 Diagrama de Tempo ou de Temporização	40
1.4.14 Síntese Geral dos Diagramas	41
1.5 Ferramentas CASE Baseadas na Linguagem UML.....	41
Capítulo 2 ■ Orientação a Objetos	43
2.1 Classificação, Abstração e Instanciação.....	43
2.2 Classes de Objetos	44
2.3 Atributos ou Propriedades	45
2.4 Métodos, Operações ou Comportamentos	46
2.5 Visibilidade	47

2.6 Herança	48
2.6.1 Herança Múltipla	49
2.7 Polimorfismo	50
Capítulo 3 ■ Diagrama de Casos de Uso	52
3.1 Atores	53
3.2 Casos de Uso	53
3.3 Documentação de Casos de Uso	55
3.4 Associações	57
3.5 Generalização/Especialização	58
3.6 Inclusão	61
3.7 Extensão	63
3.8 Restrições em Associações de Extensão	65
3.9 Pontos de Extensão	66
3.10 Multiplicidade no Diagrama de Casos de Uso	68
3.11 Estereótipos	68
3.12 Fronteira de Sistema	69
3.13 Exemplo de Diagrama de Casos de Uso – Sistema de Controle Bancário	69
3.14 Documentação do Diagrama de Casos de Uso do Sistema de Controle Bancário	72
3.14.1 Atores que Interagem com o Sistema	72
3.14.2 Documentação do Caso de Uso Abrir Conta Especial	73
3.14.3 Documentação do Caso de Uso Abrir Conta Poupança	73
3.14.4 Documentação do Caso de Uso Manter Clientes	74
3.14.5 Documentação do Caso de Uso Emitir Saldo	74
3.14.6 Documentação do Caso de Uso Emitir Extrato	75
3.14.7 Documentação do Caso de Uso Realizar Depósito	76
3.14.8 Documentação do Caso de Uso Realizar Saque	76
3.14.9 Documentação do Caso de Uso Registrar Movimento	77
3.15 Exemplo de Diagrama de Casos de Uso – Sistema de Videolocadora	78
3.16 Exemplo de Diagrama de Casos de Uso – Sistema de Telefone Celular	79
3.17 Exemplo de Diagrama de Casos de Uso – Sistema de Clínica Veterinária	82
3.18 Exemplo de Diagrama de Casos de Uso – Sistema de Controle de Advocacia	84
3.19 Exercícios Propostos	87
3.19.1 Sistema de Controle de Cinema	87
3.19.2 Sistema de Controle de Clube Social	87
3.19.3 Sistema de Locação de Veículos	88
3.19.4 Sistema para Controle de Leilão Via Internet	89
3.19.5 Sistema de Controle de Hotelaria	89
3.20 Resolução dos Exercícios	90
3.20.1 Sistema de Controle de Cinema	90
3.20.2 Sistema de Controle de Clube Social	91
3.20.3 Sistema de Locação de Veículos	93
3.20.4 Sistema para Controle de Leilão Via Internet	95
3.20.5 Sistema de Controle de Hotelaria	98

Capítulo 4 ■ Diagrama de Classes	101
4.1 Atributos e Métodos	101
4.2 Relacionamentos ou Associações	106
4.2.1 Associação Unária ou Reflexiva	107
4.2.2 Associação Binária	109
4.2.3 Associação Ternária ou N-ária	110
4.2.4 Agregação	111
4.2.5 Composição	112
4.2.6 Generalização/Especialização	113
4.2.7 Classe Associativa	115
4.2.8 Dependência	116
4.2.9 Realização	117
4.3 Portas	117
4.4 Interfaces	118
4.4.1 Interfaces Fornecidas	118
4.4.2 Interfaces Requeridas	119
4.5 Restrições	120
4.6 Estereótipos do Diagrama de Classes	126
4.6.1 Estereótipo <<entity>>	126
4.6.2 Estereótipo <<boundary>>	128
4.6.3 Estereótipo <<control>>	128
4.6.4 Estereótipos para Projeto Navegacional	129
4.6.5 Estereótipo <<enumeration>>	131
4.7 Exemplo de Diagrama de Classes – Sistema de Controle Bancário	131
4.8 Exemplo de Diagrama de Classes – Sistema de Videolocadora	141
4.9 Exemplo de Diagrama de Classes – Sistema de Telefone Celular	145
4.10 Exemplo de Diagrama de Classes – Sistema de Clínica Veterinária	149
4.11 Exemplo de Diagrama de Classes – Sistema de Controle de Advocacia	151
4.12 Persistência e Mapeamento de Classes em Tabelas	155
4.12.1 Estereótipo Table	156
4.12.2 Associações e Chaves Estrangeiras	157
4.13 Exercícios Propostos	167
4.13.1 Sistema de Controle de Cinema	167
4.13.2 Sistema de Controle de Clube Social	168
4.13.3 Sistema de Locação de Veículos	169
4.13.4 Sistema para Controle de Leilão Via Internet	169
4.13.5 Sistema de Controle de Hotelaria	170
4.14 Solução dos Exercícios	170
4.14.1 Sistema de Controle de Cinema	170
4.14.2 Sistema de Controle de Clube Social	172
4.14.3 Sistema de Locação de Veículos	174
4.14.4 Sistema para Controle de Leilão Via Internet	176
4.14.5 Sistema de Controle de Hotelaria	179

Capítulo 5 ■ Diagrama de Objetos	183
5.1 Objeto	183
5.2 Vínculos	184
5.3 Dependência com Estereótipo <<instantiate>>	185
5.4 Exemplo de Diagrama de Objetos	185
Capítulo 6 ■ Diagrama de Pacotes	187
6.1 Pacotes	187
6.2 Dependência	189
6.3 Pacotes Contendo Pacotes	190
6.4 Estereótipos Aplicados a Pacotes	191
Capítulo 7 ■ Diagrama de Sequência	192
7.1 Atores	193
7.2 Lifelines	193
7.3 Linha de Vida	195
7.4 Foco de Controle ou Ativação	195
7.5 Mensagens ou Estímulos	196
7.6 Mensagens de retorno	199
7.7 Autochamadas ou Autodelegações	200
7.8 Detalhes de Tempo	200
7.9 Mensagens Perdidas e Mensagens Encontradas	201
7.10 Portas	202
7.11 Fragmentos de Interação	203
7.12 Usos de Interação (Ocorrências de Interação antes da UML 2.1.1)	204
7.13 Portões (Gates)	206
7.14 Fragmentos Combinados e Operadores de Interação	207
7.15 Invariante de Estado (StateInvariant)	213
7.16 Exemplos de Diagramas de Sequência	215
7.16.1 Exemplo de Diagrama de Sequência – Abrir Conta Comum – Modelo Preliminar	215
7.16.2 Exemplo de Diagrama de Sequência – Abrir Conta Comum – Modelo Detalhado	216
7.16.3 Exemplo de Diagrama de Sequência – Realizar Depósito	218
7.16.4 Exemplo de Diagrama de Sequência – Emitir Extrato	219
7.17 Exercícios Propostos	221
7.17.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	221
7.17.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	221
7.17.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	221
7.17.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	222
7.17.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias	222
7.18 Solução dos Exercícios	223
7.18.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	223
7.18.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	224
7.18.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	225
7.18.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	226
7.18.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias	228

Capítulo 8 ■ Diagrama de Comunicação	231
8.1 Lifelines	231
8.2 Vínculos	232
8.3 Mensagens	233
8.4 Autochamada	233
8.5 Atores	234
8.6 Exemplo de diagrama de comunicação – Processo de Emissão de Saldo	235
8.7 Condições de Guarda e Iterações	236
8.8 Exercícios Propostos	237
8.9 Solução dos Exercícios	238
8.9.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	238
8.9.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	239
8.9.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	240
8.9.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	241
Capítulo 9 ■ Diagrama de Máquina de Estados	242
9.1 Estado	242
9.1.1 Estado Simples	243
9.2 Transições	243
9.3 Estado Inicial	244
9.4 Estado Final	244
9.5 Atividades internas	246
9.6 Transições Internas	247
9.7 Autotransições	248
9.8 Pseudoestado de Escolha	249
9.9 Barra de Bifurcação/União	251
9.10 Estados Compostos	255
9.11 Pseudoestado de História	256
9.12 Estados Compostos Ortogonais	257
9.13 Estado de Sincronismo	258
9.14 Estado de Submáquina	259
9.15 Pseudoestado de Junção	260
9.16 Pseudoestado de Ponto de Entrada e Pseudoestado de Ponto de Saída	261
9.17 Pseudoestado de Término	263
9.18 Exemplo de Diagrama de Máquina de Estados – Emitir Extrato	263
9.19 Exemplo de Diagrama de Máquina de Estados – Realizar Depósito	264
9.20 Exemplo de Diagrama de Máquina de Estados – Realizar Saque	265
9.21 Exemplo de Diagrama de Máquina de Estados – Encerrar Conta	266
9.22 Exercícios Propostos	267
9.22.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	267
9.22.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	268
9.22.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	268
9.22.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	268
9.22.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias	269

9.23 Solução dos Exercícios	270
9.23.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	270
9.23.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	270
9.23.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	272
9.23.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	273
9.23.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias	275
Capítulo 10 ■ Diagrama de Atividade	277
10.1 Atividade	278
10.2 Nó de Ação	278
10.3 Fluxo de Controle	279
10.4 Nó Inicial	280
10.5 Nó de Final de Atividade	280
10.6 Nó de Decisão	280
10.7 Exemplo Simples de Diagrama de Atividade	281
10.8 Nó de Bifurcação/União	282
10.9 Final de Fluxo	283
10.10 Fluxo de Objetos	284
10.11 Nó de Objeto	284
10.12 Alfinetes (Pins)	284
10.13 Nó de Parâmetro de Atividade	285
10.14 Exceções	286
10.15 Ação de Envio de Sinal (Ação de Objeto de Envio na versão 2.0 anterior)	286
10.16 Ação de Evento de Aceitação	286
10.17 Ação de Evento de Tempo de Aceitação	287
10.18 Nó de Buffer Central	288
10.19 Nó de Repositório de Dados (Data Store Node)	288
10.20 Conectores	288
10.21 Ação de Chamada de Comportamento	289
10.22 Ação de Chamada de Operação	290
10.23 Partição de Atividade	290
10.24 Região de Atividade Interrompível	291
10.25 Nó de Atividade Estruturada	292
10.26 Região de Expansão	293
10.27 Exemplo de Diagrama de Atividade – Emitir Extrato	294
10.28 Exemplo de Diagrama de Atividade – Realizar Depósito	296
10.29 Exemplo de Diagrama de Atividade – Realizar Saque	297
10.30 Exemplo de Diagrama de Atividade – Abrir Conta Comum	298
10.31 Exemplo de Diagrama de Atividade – Encerrar Conta	300
10.32 Exercícios Propostos	302
10.32.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	302
10.32.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	302
10.32.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	302
10.32.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	303
10.32.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias	304
10.33 Solução dos Exercícios	304
10.33.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos	304
10.33.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade	305
10.33.3 Sistema de Locação de Veículos – Processo de Locação de Veículo	307

10.33.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão	309
10.33.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias	311
Capítulo 11 ■ Diagrama de Visão Geral de Interação	313
11.1 Exemplo de Diagrama de Visão Geral de Interação – Processo Geral de Conclusão de Pedido – Sistema de Livraria Digital	315
11.2 Exercícios Propostos	316
11.2.1 Sistema de Controle de Clube Social – Processo Geral de Associação	316
11.2.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estadia	317
11.3 Solução dos Exercícios	317
11.3.1 Sistema de Controle de Clube Social – Processo Geral de Associação	317
11.3.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estadia	318
Capítulo 12 ■ Diagrama de Componentes	320
12.1 Componente	320
12.2 Interfaces Fornecidas e Requeridas	322
12.3 Classes e Componentes Internos	323
12.3.1 Portas	324
12.4 Exemplo de Diagrama de Componentes – Sistema de Controle Bancário	324
12.5 Exercícios Propostos	326
12.5.1 Sistema de Controle de Cinema	326
12.5.2 Sistema de Controle de Clube Social	327
12.5.3 Sistema de Locação de Veículos	327
12.5.4 Sistema para Controle de Leilão Via Internet	327
12.5.5 Sistema de Controle de Hotelaria	328
12.6 Solução dos Exercícios	328
12.6.1 Sistema de Controle de Cinema	328
12.6.2 Sistema de Controle de Clube Social	329
12.6.3 Sistema de Locação de Veículos	330
12.6.4 Sistema para Controle de Leilão Via Internet	330
12.6.5 Sistema de Controle de Hotelaria	332
Capítulo 13 ■ Diagrama de Implantação	334
13.1 Nós	334
13.2 Estereótipos	336
13.3 Associações	337
13.4 Exemplo de Diagrama de Implantação	337
13.5 Artefatos	339
13.6 Especificação de Implantação	340
13.7 Exemplo de Diagrama de Implantação contendo Artefatos	341
13.8 Nós Contendo Pacotes	342
13.9 Exercícios Propostos	342
13.9.1 Sistema para Controle de Leilão Via Internet	342
13.10 Solução dos Exercícios	343
13.10.1 Sistema para Controle de Leilão Via Internet	343
Capítulo 14 ■ Diagrama de Estrutura Composta	345
14.1 Colaborações	345
14.2 Papéis	346

14.3 Ocorrência de Colaboração 348

14.4 Portas 349

14.5 Propriedades e Partes 350

Capítulo 15 ■ Diagrama de Tempo ou de Temporização 352

Capítulo 16 ■ Estudo de Caso – Sistema de Pizzaria Online – PizzaNet 354

16.1 Descrição do Problema 354

16.2 Solução do Problema 359

16.2.1 Diagramas de Casos de Uso 359

16.2.2 Documentação dos Diagramas de Casos de Uso da PizzaNet 364

16.2.3 Diagrama de Classes – Modelo de Domínio 375

16.2.4 Diagrama de Objetos 387

16.2.5 Diagrama de Pacotes da PizzaNet 388

16.2.6 Diagramas de Sequência da PizzaNet 388

16.2.7 Diagrama de Comunicação Escolher Pizza 409

16.2.8 Diagramas de Máquinas de Estados da PizzaNet 410

16.2.9 Diagramas de Atividade da PizzaNet 429

16.2.10 Diagrama de Visão Geral de Interação – Realizar Pedido 457

16.2.11 Diagrama de Componentes da PizzaNet 458

16.2.12 Diagrama de Implantação da PizzaNet 459

Capítulo 17 ■ A UML 2 462

17.1 Conceitos Básicos: Modelos, Metamodelos e Metaclasses 462

17.1.1 Metaclasses Utilizadas para a Modelagem de Casos de Uso em UML 463

17.1.2 Metaclasses Classifier 464

17.1.3 Pacote CommonBehaviors, Subpacote BasicBehaviors e Metaclasses BehavioredClassifier 464

17.1.4 Pacotes Classes e Kernel e Metaclasses NameSpace, NamedElement, DirectedRelationship, Constraint e RedefinableElement 465

17.2 A Arquitetura da Linguagem 466

17.2.1 Princípios de Projeto da UML 2 466

17.2.2 A Infraestrutura da UML 2 467

17.2.3 O Pacote Núcleo da Biblioteca de Infraestrutura 468

17.2.4 Perfis 470

17.2.5 Alinhamento Arquitetural entre a UML e a MOF 470

17.2.6 Reutilizando a Infraestrutura 471

17.2.7 O Pacote Central (Kernel) 471

17.2.8 Camadas do Metamodelo 472

17.2.9 A Hierarquia de metamodelo de quatro camadas 472

17.2.10 Metamodelagem 473

17.3 Adaptando e Estendendo a UML 474

17.3.1 Criação de Perfis 474

17.3.2 Estendendo o Metamodelo 476

17.4 A Superestrutura da UML 2 477

Índice remissivo 479

Agradecimentos

Agradeço primeiramente ao professor Carlos Emilio Padilla Severo, pelas muitas ideias e esclarecimentos que foram produzidos durante nossas conversas.

Agradeço também aos meus alunos das disciplinas de Engenharia de Software e afins do curso de Licenciatura Plena em Informática da Universidade Federal de Mato Grosso, UFMT, Campus de Rondonópolis; aos meus alunos do curso de Sistemas de Informação do CESUR/FACSUL, Faculdade do Sul de Mato Grosso, no qual lecionei antes de ingressar na UFMT, e aos meus alunos do curso de pós-graduação em Processo e Desenvolvimento de Software da Faculdade Exponencial, FIE, de Chapecó, em Santa Catarina.

Sobre o Autor

Gilleanes Thorwald Araujo Guedes é mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS) e bacharel em Informática pela Universidade da Região da Campanha (URCAMP). É professor de Engenharia de Software no curso de Licenciatura Plena em Informática na Universidade Federal de Mato Grosso (UFMT), estando atualmente cursando o doutorado em Ciência da Computação na Universidade Federal do Rio Grande do Sul. Já ministrou diversas palestras e cursos sobre UML em eventos científicos, cursos técnicos e cursos de pós-graduação “lato sensu”. É autor dos livros UML – Uma abordagem prática, UML 2 – Guia de consulta rápida e UML 2 – Guia prático, publicados pela Novatec Editora. Pode ser contatado pelo e-mail gtag@novatec.com.br.

Prefácio

A UML (Unified Modeling Language – Linguagem de Modelagem Unificada) tornou-se, nos últimos anos, a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software. Em decorrência disso, existe hoje uma grande valorização e procura no mercado por profissionais que dominem essa linguagem.

O objetivo deste livro é ensinar ao leitor como modelar software por meio dos diversos diagramas que compõem a UML. No entanto, é importante destacar que a UML é uma linguagem de modelagem totalmente independente, não estando vinculada a nenhum processo de desenvolvimento específico e menos ainda a qualquer linguagem de programação.

Apesar de a UML oferecer um grande número de diagramas que enfoquem tanto características estruturais quanto comportamentais de um software, o leitor não deve se sentir obrigado a utilizar todos os diagramas propostos na modelagem de seus sistemas, pois cada um deles tem uma função específica e, algumas vezes, alguns deles não são necessários em determinadas situações ou domínios.

Esta obra foi revisada, atualizada e ampliada tomando como base o livro UML – Uma Abordagem Prática, que teve sua primeira edição lançada no início de 2004. Contudo, nessa primeira obra utilizamos principalmente a UML em sua versão 1.5, enquanto neste novo livro empregamos totalmente as notações definidas na UML 2, que apresenta grandes inovações com relação às versões anteriores. Assim, embora alguns exemplos sejam parecidos com os do primeiro livro, estes foram todos revisados e atualizados sempre que isso se mostrou necessário. Além disso, o livro contém uma grande quantidade de novos exemplos e propõe exercícios igualmente inéditos, sendo que todos os capítulos utilizam a notação da UML 2 e demonstram ainda os novos componentes e características acrescidos a cada diagrama. Alguns desses capítulos se referem a diagramas novos, que só passaram a existir ou se tornaram independentes de outros com o advento da UML 2. O estudo de caso apresentado ao final do livro também é totalmente inédito, onde modelamos um sistema para controle de pizzeria online.

O livro está estruturado da seguinte forma:

O capítulo 1 apresenta uma explanação a respeito da necessidade de se modelar software, além de introduzir a UML, destacando em linhas gerais as funções de cada diagrama.

O capítulo 2 enfoca o paradigma de orientação a objetos, uma vez que a UML é uma linguagem baseada nesse paradigma e utilizada principalmente para a modelagem de softwares orientados a objetos.

Os capítulos 3 a 15 abordam, respectivamente, os diagramas de casos de uso, de classes, objetos, pacotes, sequência, comunicação, máquina de estados, atividade, visão geral de interação, componentes, implantação, estrutura composta e tempo. Em cada um desses capítulos procuramos descrever a função de cada diagrama, detalhando seus componentes e apresentando exemplos de como utilizar cada diagrama. Ao final da maioria dos capítulos são sugeridos alguns exercícios para que o leitor possa praticar seu conhecimento. Todos os exercícios encontram-se resolvidos e explicados no final de seus respectivos capítulos.

Ao longo dos capítulos 3 a 14 foi modelado um sistema de controle bancário como ilustração e, ao longo dos exercícios, foram parcialmente modelados cinco sistemas relativamente simples. O capítulo 16 apresenta um estudo de caso referente a um sistema maior e mais complexo, no qual modelamos um sistema para controle de pizzeria online, onde os pedidos dos clientes poderão ser feitos pela internet.

Finalmente, no último capítulo enfocamos a arquitetura da linguagem, discorrendo sobre a infraestrutura e superestrutura da UML 2.



CAPÍTULO 1

Introdução à UML

A UML – Unified Modeling Language ou Linguagem de Modelagem Unificada – é uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objetos. É uma linguagem de modelagem de propósito geral que pode ser aplicada a todos os domínios de aplicação. Essa linguagem tornou-se, nos últimos anos, a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software.

Deve ficar bem claro, porém, que a UML não é uma linguagem de programação, e sim uma linguagem de modelagem, uma notação, cujo objetivo é auxiliar os engenheiros de software a definirem as características do sistema, tais como seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos e até mesmo suas necessidades físicas em relação ao equipamento sobre o qual o sistema deverá ser implantado. Tais características podem ser definidas por meio da UML antes do software começar a ser realmente desenvolvido. Além disso, cumpre destacar que a UML não é um processo de desenvolvimento de software e tampouco está ligada a um de forma exclusiva, sendo totalmente independente, podendo ser utilizada por muitos processos de desenvolvimento diferentes ou mesmo da forma que o engenheiro considerar mais adequada.

1.1 Breve Histórico da UML

A UML surgiu da união de três métodos de modelagem: o método de Booch, o método OMT (Object Modeling Technique) de Jacobson, e o método OOSE (Object-Oriented Software Engineering) de Rumbaugh. Estes eram, até meados da década de 1990, os métodos de modelagem orientada a objetos mais populares entre os profissionais da área de desenvolvimento de software. A união desses métodos contou com o amplo apoio da Rational Software, que a incentivou e financiou.

O esforço inicial do projeto começou com a união do método de Booch ao OMT de Jacobson, o que resultou no lançamento do Método Unificado no final de 1995. Logo em seguida, Rumbaugh juntou-se a Booch e Jacobson na

Rational Software, e seu método OOSE começou também a ser incorporado à nova metodologia. O trabalho de Booch, Jacobson e Rumbaugh, conhecidos popularmente como “Os Três Amigos”, resultou no lançamento, em 1996, da primeira versão da UML propriamente dita.

Tão logo a primeira versão foi lançada, muitas empresas atuantes na área de modelagem e desenvolvimento de software passaram a contribuir para o projeto, fornecendo sugestões para melhorar e ampliar a linguagem. Finalmente, a UML foi adotada, em 1997, pela OMG (Object Management Group ou Grupo de Gerenciamento de Objetos), como uma linguagem-padrão de modelagem.

A versão 2.0 da linguagem foi oficialmente lançada em julho de 2005, encontrando-se esta atualmente na versão 2.3 beta. A documentação oficial da UML pode ser consultada no site da OMG em www.omg.org ou mais exatamente em www.uml.org.

1.2 Por Que Modelar Software?

Qual a real necessidade de se modelar um software? Muitos “profissionais” podem afirmar que conseguem determinar todas as necessidades de um sistema de informação de cabeça, e que sempre trabalharam assim. Qual a real necessidade de se projetar uma casa? Um pedreiro experiente não é capaz de construí-la sem um projeto? Isso pode ser verdade, mas a questão é muito mais ampla, envolvendo fatores extremamente complexos, como levantamento e análise de requisitos, prototipação, tamanho do projeto, complexidade, prazos, custos, documentação, manutenção e reusabilidade, entre outros.

Existe uma diferença gritante entre construir uma pequena casa e construir um prédio de vários andares. Obviamente, para se construir um edifício é necessário, em primeiro lugar, desenvolver um projeto muito bem-elaborado, cujos cálculos têm de estar extremamente corretos e precisos. Além disso, é preciso fornecer uma estimativa de custos, determinar em quanto tempo a construção estará concluída, avaliar a quantidade de profissionais necessária à execução da obra, especificar a quantidade de material a ser adquirida para a construção, escolher o local onde o prédio será erguido etc. Grandes projetos não podem ser modelados de cabeça, nem mesmo a maioria dos pequenos projetos pode sê-lo, exceto, talvez, aqueles extremamente simples.

Na realidade, por mais simples que seja, todo e qualquer sistema deve ser modelado antes de se iniciar sua implementação, entre outras coisas, porque os sistemas de informação frequentemente costumam ter a propriedade de “crescer”, isto é, aumentar em tamanho, complexidade e abrangência. Muitos profissionais costumam afirmar que sistemas de informação são “vivos”, porque nunca estão completamente finalizados. Na verdade, o termo correto seria “dinâmicos”, pois

normalmente os sistemas de informação estão em constante mudança. Tais mudanças são devidas a diversos fatores, como, por exemplo:

- Os clientes desejam constantemente modificações ou melhorias no sistema.
- O mercado está sempre mudando, o que força a adoção de novas estratégias por parte das empresas e, conseqüentemente, de seus sistemas.
- O governo seguidamente promulga novas leis e cria novos impostos e alíquotas ou, ainda, modifica as leis, os impostos e alíquotas já existentes, o que acarreta a manutenção no software.

Assim, um sistema de informação precisa ter uma documentação extremamente detalhada, precisa e atualizada para que possa ser mantido com facilidade, rapidez e correção, sem produzir novos erros ao corrigir os antigos. Modelar um sistema é uma forma bastante eficiente de documentá-lo, mas a modelagem não serve apenas para isso: a documentação é apenas uma das vantagens fornecidas pela modelagem. Existem muitas outras que serão discutidas nas próximas seções.

1.2.1 Modelo de Software – Uma Definição

A modelagem de um software implica em criar modelos de software, mas o que é realmente um modelo de software? Um modelo de software captura uma visão de um sistema físico, é uma abstração do sistema com um certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que é considerado irrelevante. Assim um modelo descreve completamente aqueles aspectos do sistema físico que são relevantes ao propósito do modelo, no nível apropriado de detalhe.

Dessa forma, um modelo de casos de uso fornecerá uma visão dos requisitos necessários ao sistema, identificando as funcionalidades do software e os atores que poderão utilizá-las, não se preocupando em detalhar nada além disso. Já um modelo conceitual irá identificar as classes relacionadas ao domínio do problema, sem detalhar seus métodos, enquanto um modelo de domínio ampliará o modelo conceitual, incluindo informações relativas à solução do problema, incluindo, entre outras coisas, os métodos necessários a essa solução.

1.2.2 Levantamento e Análise de Requisitos

Uma das primeiras fases de um processo de desenvolvimento de software consiste no Levantamento de Requisitos. As outras etapas, sugeridas por muitos autores, são: Análise de Requisitos, Projeto, que se constitui na principal fase da modelagem, Codificação, Testes e Implantação. Dependendo do método/processo adotado, essas etapas ganham, por vezes, nomenclaturas diferentes, podendo algumas delas ser condensadas em uma etapa única, ou uma etapa pode ser

dividida em duas ou mais etapas. Se tomarmos como exemplo o Processo Unificado (Unified Process), um método de desenvolvimento de software, veremos que este se divide em quatro fases: Concepção, onde é feito o levantamento de requisitos; Elaboração, onde é feita a análise dos requisitos e o projeto do software; Construção, onde o software é implementado e testado; e Transição, onde o software será implantado. As fases de Elaboração e Construção ocorrem, sempre que possível, em ciclos iterativos, dessa forma, sempre que um ciclo é completado pela fase de Construção, volta-se à fase de Elaboração para tratar do ciclo seguinte, até todo o software ser finalizado.

As etapas de levantamento e análise de requisitos trabalham com o domínio do problema e tentam determinar “o que” o software deve fazer e se é realmente possível desenvolver o software solicitado. Na etapa de levantamento de requisitos, o engenheiro de software busca compreender as necessidades do usuário e o que ele deseja que o sistema a ser desenvolvido realize. Isso é feito sobretudo por meio de entrevistas, nas quais o engenheiro tenta compreender como funciona atualmente o processo a ser informatizado e quais serviços o cliente precisa que o software forneça.

Devem ser realizadas tantas entrevistas quantas forem necessárias para que as necessidades do usuário sejam bem-compreendidas. Durante as entrevistas, o engenheiro deve auxiliar o cliente a definir quais informações deverão ser produzidas, quais deverão ser fornecidas e qual o nível de desempenho exigido do software.

Um dos principais problemas enfrentados na fase de levantamento de requisitos é o de comunicação. A comunicação constitui-se em um dos maiores desafios da engenharia de software, caracterizando-se pela dificuldade em conseguir compreender um conjunto de conceitos vagos, abstratos e difusos que representam as necessidades e os desejos dos clientes e transformá-los em conceitos concretos e inteligíveis.

A fase de levantamento de requisitos deve identificar dois tipos de requisitos: os funcionais e os não-funcionais. Os requisitos funcionais correspondem ao que o cliente quer que o sistema realize, ou seja, as funcionalidades do software. Já os requisitos não-funcionais correspondem às restrições, condições, consistências, validações que devem ser levadas a efeito sobre os requisitos funcionais. Por exemplo, em um sistema bancário deve ser oferecida a opção de abrir novas contas correntes, o que é um requisito funcional. Já determinar que somente pessoas maiores de idade possam abrir contas corrente é um requisito não-funcional.

Podem existir diversos tipos de requisitos não-funcionais, como de usabilidade, desempenho, confiabilidade, segurança ou interface. Alguns requisitos não-funcionais identificam regras de negócio, ou seja, as políticas, normas e condições estabelecidas pela empresa que devem ser seguidas na execução de

uma funcionalidade. Por exemplo, estabelecer que depois de abrir uma conta é necessário depositar um valor mínimo inicial é uma regra de negócio adotada por um determinado banco e que não necessariamente é seguida por outras instituições bancárias. Outro exemplo de regra de negócio seria determinar que, em um sistema de videolocadora, só se poderia realizar uma nova locação para um sócio depois de ele ter devolvido as cópias locadas anteriormente.

Logo após o levantamento de requisitos, passa-se à fase em que as necessidades apresentadas pelo cliente são analisadas. Essa etapa é conhecida como análise de requisitos. Aqui o engenheiro examina os requisitos enunciados pelos usuários, verificando se estes foram especificados corretamente e se foram realmente bem-compreendidos. A partir da etapa de análise de requisitos são determinadas as reais necessidades do sistema de informação.

A grande questão é: como saber se as necessidades dos usuários foram realmente bem-compreendidas? Um dos objetivos da análise de requisitos consiste em determinar se as necessidades dos usuários foram entendidas de maneira correta, verificando se alguma questão deixou de ser abordada, determinando se algum requisito foi especificado incorretamente ou se algum conceito precisa ser melhor explicado. Durante a análise de requisitos, uma linguagem de modelagem auxilia a levantar questões que não foram concebidas durante as entrevistas iniciais. Tais questões devem ser sanadas o quanto antes, para que o projeto do software não tenha que sofrer modificações quando seu desenvolvimento já estiver em andamento, o que pode causar significativos atrasos no desenvolvimento do software, sendo por vezes necessário remodelar por completo o projeto.

Além daquele concernente à comunicação, outro grande problema encontrado durante as entrevistas consiste no fato de que, na maioria das vezes, os usuários não têm realmente certeza do que querem e não conseguem enxergar as reais potencialidades de um sistema de informação. Em geral, os engenheiros de software precisam sugerir inúmeras características e funções do sistema que o cliente não sabia como formular ou sequer havia imaginado. Na realidade, na maior parte das vezes, esses profissionais precisam reestruturar o modo como as informações são geridas e utilizadas pela empresa e apresentar maneiras de combiná-las e apresentá-las de maneira que possam ser melhor aproveitadas pelos usuários.

Em muitos casos é realmente isso o que os clientes esperam dos engenheiros de software, porém, em outros, os engenheiros encontram fortes resistências a qualquer mudança na forma como a empresa manipula suas informações, fazendo-se necessário um significativo esforço para provar ao cliente que as modificações sugeridas permitirão um melhor desempenho do software, além

de ser útil para a própria empresa, obviamente. Na realidade, nesse último caso é fundamental trabalhar bastante o aspecto social da implantação de um sistema informatizado na empresa, pois muitas vezes a resistência não é tanto por parte da gerência, mas pelos usuários finais, que serão obrigados a mudar a forma como estavam acostumados a trabalhar e aprender a utilizar uma nova tecnologia.

1.2.3 Prototipação

A prototipação é uma técnica bastante popular e de fácil aplicação. Essa técnica consiste em desenvolver rapidamente um “rascunho” do que seria o sistema de informação quando ele estivesse finalizado. Um protótipo normalmente apresenta pouco mais do que a interface do software a ser desenvolvido, ilustrando como as informações seriam inseridas e recuperadas no sistema, apresentando alguns exemplos com dados fictícios de quais seriam os resultados apresentados pelo software, principalmente em forma de relatórios. A utilização de um protótipo pode, assim, evitar que, após meses ou até anos de desenvolvimento, descubra-se, ao implantar o sistema, que o software não atende completamente às necessidades do cliente devido, sobretudo, a falhas de comunicação durante as entrevistas iniciais.

Hoje em dia, é possível desenvolver protótipos com extrema rapidez e facilidade, por meio da utilização de ferramentas conhecidas como RAD (Rapid Application Development ou Desenvolvimento Rápido de Aplicações). Essas ferramentas são encontradas na maioria dos ambientes de desenvolvimento das linguagens de programação atuais, como NetBeans, Delphi, Visual Basic ou C++ Builder, entre outras. Essas ferramentas disponibilizam ambientes de desenvolvimento que permitem a construção de interfaces de forma muito rápida, além de permitirem também modificar tais interfaces de maneira igualmente veloz, na maioria das vezes sem a necessidade de alterar qualquer código porventura já escrito.

As ferramentas RAD permitem a criação de formulários e a inserção de componentes nos mesmos, de uma forma muito simples, rápida e fácil, bastando ao desenvolvedor selecionar o componente (botões, caixas de texto, labels, combos etc.) em uma barra de ferramentas e clicar com o mouse sobre o formulário. Alternativamente, o usuário pode clicar duas vezes sobre o componente desejado, fazendo com que um componente do tipo selecionado surja no centro do formulário. Além disso, tais ferramentas permitem ao usuário mudar a posição dos componentes depois de terem sido colocados no formulário simplesmente selecionando o componente com o mouse e o arrastando para a posição desejada.

Esse tipo de ferramenta é extremamente útil no desenvolvimento de protótipos pela facilidade de produzir e modificar as interfaces. Assim, depois de determinar quais as modificações necessárias ao sistema de informação após o

protótipo ter sido apresentado aos usuários, pode-se modificar a interface do protótipo de acordo com as novas especificações e reapresentá-lo ao cliente de forma muito rápida.

Seguindo esse raciocínio, a etapa de análise de requisitos deve, obrigatoriamente, produzir um protótipo para demonstrar como se apresentará e comportará o sistema em essência, bem como quais informações deverão ser inseridas no sistema e que tipo de informações deverão ser fornecidas pelo software. Um protótipo é de extrema importância durante as primeiras fases de engenharia de um sistema de informação. Por meio da ilustração que um protótipo pode apresentar, a maioria das dúvidas e erros de especificação pode ser sanada, devido ao fato de um protótipo demonstrar visualmente um exemplo de como funcionará o sistema depois de concluído, como será sua interface, de que maneira os usuários interagirão com ele, que tipo de relatórios serão fornecidos etc., facilitando a compreensão do cliente.

Apesar das grandes vantagens advindas do uso da técnica de prototipação, é necessária ainda uma ressalva: um protótipo pode induzir o cliente a acreditar que o software encontra-se em um estágio bastante avançado de desenvolvimento. Com frequência ocorre de o cliente não compreender o conceito de um protótipo. Para ele, o esboço apresentado já é o próprio sistema praticamente acabado. Por isso, muitas vezes o cliente não compreende nem aceita prazos longos, os quais considera absurdos, já que o sistema foi-lhe apresentado já funcionando, necessitando de alguns poucos ajustes. Por isso, é preciso deixar bem claro ao usuário que o software que lhe está sendo apresentado é apenas um “rascunho” do que será o sistema de informação quando estiver finalizado e que seu desenvolvimento ainda não foi realmente iniciado.

1.2.4 Prazos e Custos

Em seguida, vem a espinhosa e desagradável, porém extremamente importante, questão dos prazos e custos. Como determinar o prazo real de entrega de um software? Quantos profissionais deverão trabalhar no projeto? Qual será o custo total de desenvolvimento? Qual deverá ser o valor estipulado para produzir o sistema? Como determinar a real complexidade de desenvolvimento do software? Geralmente, após as primeiras entrevistas, os clientes estão bastante interessados em saber quanto vai lhes custar o sistema de informação e em quanto tempo eles o terão implantado e funcionando em sua empresa.

A estimativa de tempo é realmente um tópico extremamente complexo da engenharia de software. Na realidade, por melhor modelado que um sistema tenha sido, ainda assim fica difícil determinar com exatidão os prazos finais de entrega do software. Uma boa modelagem auxilia a estimar a complexidade de

desenvolvimento de um sistema, e isso, por sua vez, ajuda – e muito – a determinar o prazo final em que o software será entregue. No entanto, é preciso ter diversos sistemas de informação com níveis de dificuldade e características semelhantes ao software que está para ser construído, já previamente desenvolvidos e bem-documentados, para determinar com maior exatidão a estimativa de prazos.

Contudo, mesmo com o auxílio dessa documentação, ainda é muito difícil estipular uma data exata. O máximo que se pode conseguir é apresentar uma que seja aproximada, com base na experiência documentada de desenvolvimento de outros softwares. Assim, é recomendável acrescentar alguns meses à data de entrega, o que serve como margem de segurança para possíveis erros de estimativa.

Para poder auxiliar na estimativa de prazos e custos de um software, a documentação da empresa desenvolvedora deverá ter registros das datas de início e término de cada projeto já concluído, além do custo real de desenvolvimento que tais projetos acarretaram, envolvendo inclusive os custos com manutenção e o número de profissionais envolvidos em cada projeto. Na verdade, uma empresa de desenvolvimento de software que nunca tenha desenvolvido um sistema de informação antes e, portanto, não tenha documentação histórica de projetos anteriores, dificilmente será capaz de apresentar uma estimativa correta de prazos e custos, principalmente porque a equipe de desenvolvimento não saberá com certeza quanto tempo levará desenvolvendo o sistema, já que o tempo de desenvolvimento influencia diretamente o custo de desenvolvimento do sistema e, logicamente, o valor a ser cobrado pelo software.

Se a estimativa de prazo estiver errada, cada dia a mais de desenvolvimento do projeto acarretará prejuízos para a empresa que desenvolve o sistema, decorrentes, por exemplo, de pagamentos de salários aos profissionais envolvidos no projeto que não haviam sido previstos e desgaste dos equipamentos utilizados. Isso sem levar em conta prejuízos mais difíceis de contabilizar, como manter inúmeros profissionais ocupados em projetos que já deveriam estar concluídos, que deixam de trabalhar em novos projetos, além da insatisfação dos clientes por não receberem o produto no prazo estimado e a propaganda negativa daí decorrente.

1.2.5 Projeto

Enquanto a fase de análise trabalha com o domínio do problema, a fase de projeto trabalha com o domínio da solução, procurando estabelecer “como” o sistema fará o que foi determinado na fase de análise, ou seja, qual será a solução para o problema identificado. É na etapa de projeto que é realizada a maior parte da modelagem do software a ser desenvolvido, ou seja, é nessa etapa que é produzida a arquitetura do sistema.

A etapa de projeto toma a modelagem iniciada na fase de análise e lhe acrescenta profundos acréscimos e detalhamentos. Enquanto na análise foram identificadas as funcionalidades necessárias ao software e suas restrições, na fase de projeto será estabelecido como essas funcionalidades deverão realizar o que foi solicitado.

A fase de projeto leva em consideração os recursos tecnológicos existentes para que o problema apresentado pelo cliente possa ser solucionado. É nesse momento que será selecionada a linguagem de programação a ser utilizada, o sistema gerenciador de banco de dados a ser empregado, como será a interface final do sistema e até mesmo como o software será distribuído fisicamente na empresa, especificando o hardware necessário para a sua implantação e funcionamento correto.

1.2.6 Manutenção

Possivelmente a questão mais importante que todas as outras já enunciadas é a da manutenção. Alguns autores afirmam que muitas vezes a manutenção de um software pode representar de 40 a 60% do custo total do projeto. Alguém poderá então dizer que a modelagem é necessária para diminuir os custos com manutenção – se a modelagem estiver correta o sistema não apresentará erros e, então, não precisará sofrer manutenções.

Embora um dos objetivos de modelar um software seja realmente diminuir a necessidade de mantê-lo, a modelagem não serve apenas para isso. Na maioria dos casos, a manutenção de um software é inevitável, pois, como já foi dito, as necessidades de uma empresa são dinâmicas e mudam constantemente, o que faz surgir novas necessidades que não existiam na época em que o software foi projetado, isso sem falar nas frequentes mudanças em leis, alíquotas, impostos, taxas ou formato de notas fiscais, por exemplo. Levando isso em consideração, é bastante provável que um sistema de informação, por mais bem-modelado que esteja, precise sofrer manutenções.

Nesse caso, a modelagem não serve apenas para diminuir a necessidade de futuras manutenções, mas também para facilitar a compreensão do sistema por quem tiver que mantê-lo, já que, em geral, a manutenção de um sistema é considerada uma tarefa ingrata pelos profissionais de desenvolvimento, por normalmente exigir que estes despendam grandes esforços para compreender códigos escritos por outros cujos estilos de desenvolvimento são diferentes e que, via de regra, não se encontram mais na empresa.

Esse tipo de código é conhecido como “código alienígena” ou “software legado”. O termo refere-se a códigos que não seguem as regras atuais de desenvolvimento da empresa, não foram modelados e, por conseguinte, têm pouca ou nenhuma

documentação. Além disso, nenhum dos profissionais da equipe atual trabalhou em seu projeto inicial e, para piorar, o código já sofreu manutenções anteriores por outros profissionais que também não se encontram mais na empresa, sendo que cada um deles tinha um estilo de desenvolvimento diferente, ou seja, como se diz no meio de desenvolvimento, o código encontra-se “remendado”.

Assim, uma modelagem correta aliada a uma documentação completa e atualizada de um sistema de informação torna mais rápido o processo de manutenção e impede que erros sejam cometidos, já que é muito comum que, depois de manter uma rotina ou função de um software, outras rotinas ou funções do sistema que antes funcionavam perfeitamente passem a apresentar erros ou simplesmente deixem de funcionar. Tais erros são conhecidos como “efeitos colaterais” da manutenção.

Além disso, qualquer manutenção a ser realizada em um sistema deve ser também modelada e documentada, para não desatualizar a documentação do sistema e prejudicar futuras manutenções, já que muitas vezes uma documentação desatualizada pode ser mais prejudicial à manutenção do sistema do que nenhuma documentação.

Pode-se fornecer uma analogia de “manutenção” na vida real responsável pela produção de um efeito colateral no meio ambiente, o que não deixa de ser um sistema: muito pelo contrário, é “o” sistema. Na realidade, esse exemplo não identifica exatamente uma manutenção, e sim uma modificação em uma região. Descobri recentemente, ao assistir a uma reportagem, que a formiga saúva vinha se tornando uma praga em algumas regiões do país porque estava se reproduzindo demais. Esse crescimento desordenado era causado pelos tratores que, ao arar a terra, destruíam os formigueiros da formiga Lava-Pés, que ficam próximos à superfície, mas não afetavam os formigueiros de saúvas, por estes se encontrarem em um nível mais profundo do solo.

Entretanto, as lava-pés consumiam os ovos das saúvas, o que impedia que estas aumentassem demais. Assim, a diminuição das formigas lava-pés resultou no crescimento desordenado das saúvas. Isso é um exemplo de manutenção com efeito colateral na vida real. No caso, foi aplicada uma espécie de “manutenção”, onde modificou-se o ambiente para arar a terra e produziu-se uma praga que antes constituía-se apenas em uma espécie controlada. Se a “função” da formiga lava-pés estivesse devidamente documentada, ela não teria sido eliminada, e a saúva, por conseguinte, não teria se tornado uma praga.

1.2.7 Documentação Histórica

Finalmente, existe a questão da perspectiva histórica, ou seja, novamente a já tão falada documentação de software. Neste caso, referimo-nos à documentação histórica dos projetos anteriores já concluídos pela empresa. É por meio dessa documentação histórica que a empresa pode responder a perguntas como:

- A empresa está evoluindo?
- O processo de desenvolvimento tornou-se mais rápido?
- As metodologias hoje adotadas são superiores às práticas aplicadas anteriormente?
- A qualidade do software produzido está melhorando?

Uma empresa ou setor de desenvolvimento de software necessita de um registro detalhado de cada um de seus sistemas de informação antes desenvolvidos para poder determinar, entre outros, fatores como:

- A média de manutenções que um sistema sofre normalmente dentro de um determinado período de tempo.
- Qual a média de custo de modelagem.
- Qual a média de custo de desenvolvimento.
- Qual a média de tempo despendido até a finalização do projeto.
- Quantos profissionais são necessários envolver normalmente em um projeto.

Essas informações são computadas nos orçamentos de desenvolvimento de novos softwares e são de grande auxílio no momento de determinar prazos e custos mais próximos da realidade.

Além disso, a documentação pode ser muito útil em outra área: a Reusabilidade. Um dos grandes desejos e muitas vezes necessidades dos clientes é que o software esteja concluído o mais rápido possível. Uma das formas de agilizar o processo de desenvolvimento é a reutilização de rotinas, funções e algoritmos previamente desenvolvidos em outros sistemas. Nesse caso, a documentação correta do sistema pode auxiliar a sanar questões como:

- Onde as rotinas se encontram?
- Para que foram utilizadas?
- Em que projetos estão documentadas?
- Elas são adequadas ao software atualmente em desenvolvimento?
- Qual o nível necessário de adaptação destas rotinas para que possam ser utilizadas na construção do sistema atual?

1.3 Por que tantos Diagramas?

Por que a UML é composta por tantos diagramas? O objetivo disso é fornecer múltiplas visões do sistema a ser modelado, analisando-o e modelando-o sob diversos aspectos, procurando-se, assim, atingir a completitude da modelagem, permitindo que cada diagrama complemente os outros.

Cada diagrama da UML analisa o sistema, ou parte dele, sob uma determinada óptica. É como se o sistema fosse modelado em camadas, sendo que alguns diagramas enfocam o sistema de forma mais geral, apresentando uma visão externa do sistema, como é o objetivo do Diagrama de Casos de Uso, enquanto outros oferecem uma visão de uma camada mais profunda do software, apresentando um enfoque mais técnico ou ainda visualizando apenas uma característica específica do sistema ou um determinado processo. A utilização de diversos diagramas permite que falhas sejam descobertas, diminuindo a possibilidade da ocorrência de erros futuros.

Tomando novamente o exemplo da construção de um edifício, percebemos que ao se projetar uma construção, esta não tem apenas uma planta, mas diversas, enfocando o projeto de construção do prédio sob diferentes formas, algumas referentes ao layout dos andares, outras apresentando a planta hidráulica e outras ainda abordando a planta elétrica, por exemplo. Isso torna o projeto do edifício completo, abrangendo todas as características da construção. Da mesma maneira, os diversos diagramas fornecidos pela UML permitem analisar o sistema em diferentes níveis, podendo focar a organização estrutural do sistema, o comportamento de um processo específico, a definição de um determinado algoritmo ou até mesmo as necessidades físicas para a implantação do sistema.

1.4 Rápido Resumo dos Diagramas da UML

A seguir descreveremos rapidamente cada um dos diagramas oferecidos pela UML, destacando suas principais características.

1.4.1 Diagrama de Casos de Uso

O diagrama de casos de uso é o diagrama mais geral e informal da UML, utilizado normalmente nas fases de levantamento e análise de requisitos do sistema, embora venha a ser consultado durante todo o processo de modelagem e possa servir de base para outros diagramas. Apresenta uma linguagem simples e de fácil compreensão para que os usuários possam ter uma ideia geral de como o sistema irá se comportar. Procura identificar os atores (usuários, outros sistemas ou até mesmo algum hardware especial) que utilizarão de alguma forma

o software, bem como os serviços, ou seja, as funcionalidades que o sistema disponibilizará aos atores, conhecidas nesse diagrama como casos de uso. Veja na figura 1.1 um exemplo desse diagrama.

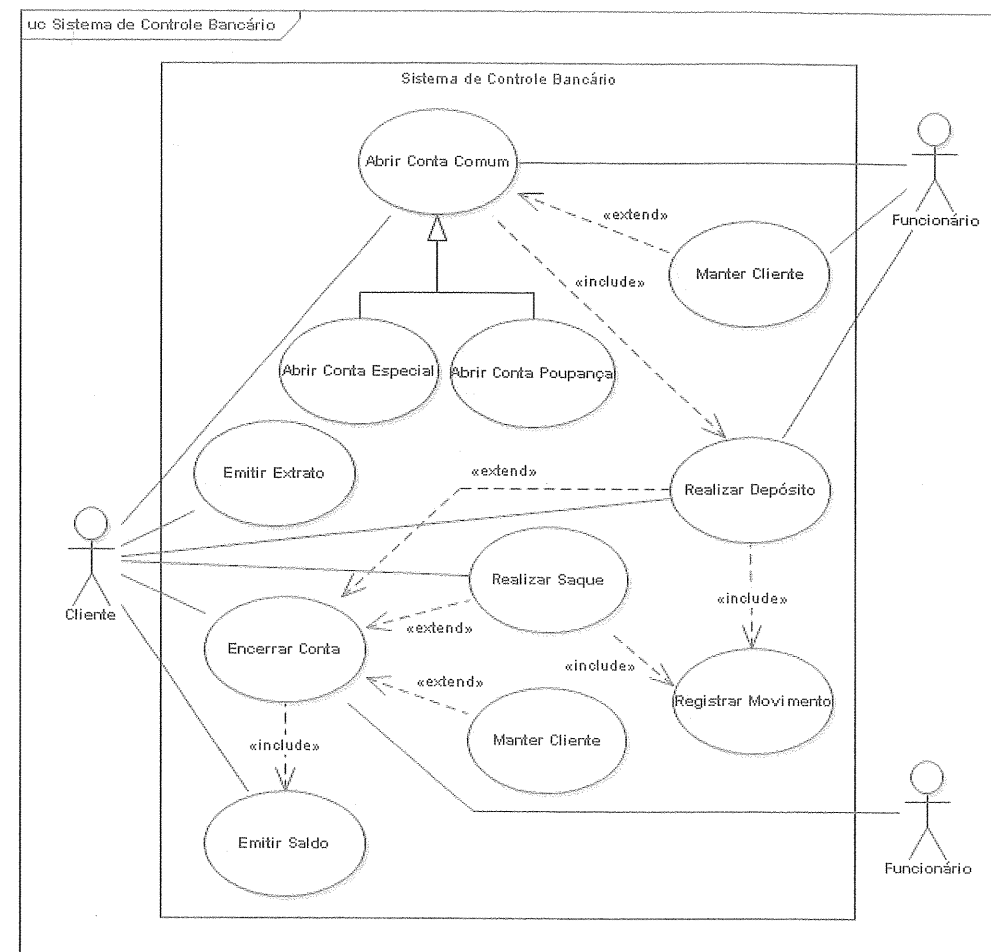


Figura 1.1 – Exemplo de Diagrama de Casos de Uso.

1.4.2 Diagrama de Classes

O diagrama de classes é provavelmente o mais utilizado e é um dos mais importantes da UML. Serve de apoio para a maioria dos demais diagramas. Como o próprio nome diz, define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos que cada classe tem, além de estabelecer como as classes se relacionam e trocam informações entre si. A figura 1.2 apresenta um exemplo desse diagrama.

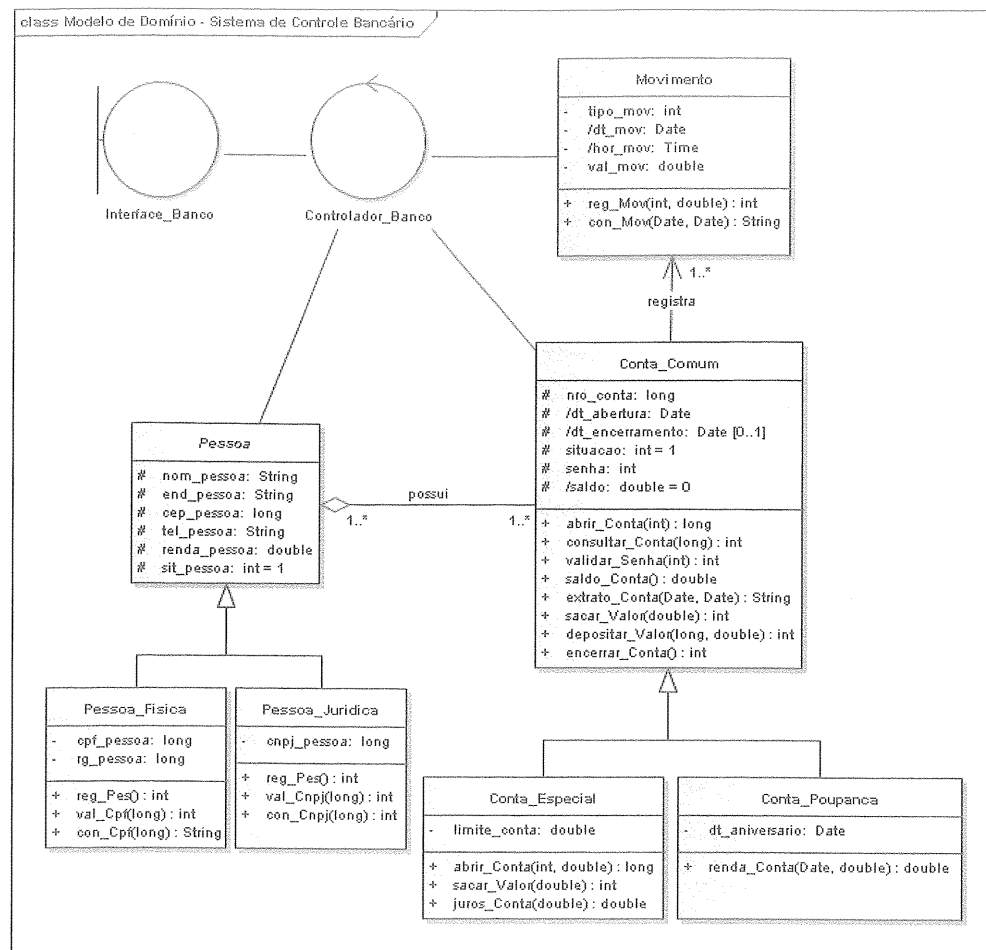


Figura 1.2 – Exemplo de Diagrama de Classes.

1.4.3 Diagrama de Objetos

O diagrama de objetos está amplamente associado ao diagrama de classes. Na verdade, o diagrama de objetos é praticamente um complemento do diagrama de classes e bastante dependente deste. O diagrama fornece uma visão dos valores armazenados pelos objetos de um diagrama de classes em um determinado momento da execução de um processo do software. A figura 1.3 apresenta um exemplo de diagrama de objetos.

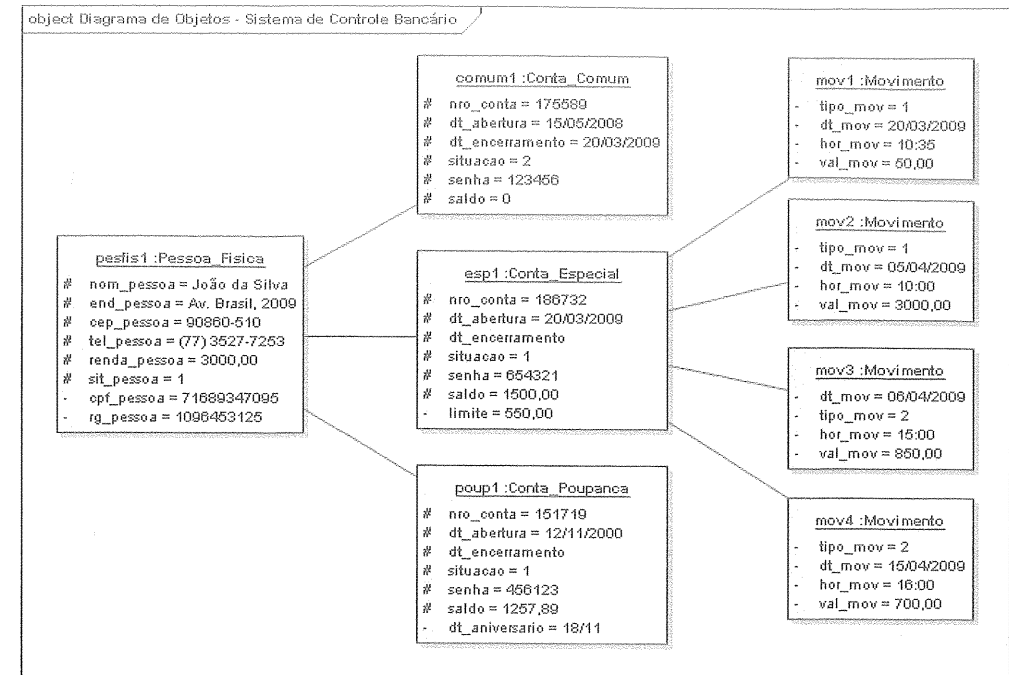


Figura 1.3 – Exemplo de Diagrama de Objetos.

1.4.4 Diagrama de Pacotes

O diagrama de pacotes é um diagrama estrutural que tem por objetivo representar os subsistemas ou submódulos englobados por um sistema de forma a determinar as partes que o compõem. Pode ser utilizado de maneira independente ou associado com outros diagramas. Esse diagrama pode ser utilizado também para auxiliar a demonstrar a arquitetura de uma linguagem, como ocorre com a própria UML ou ainda para definir as camadas de um software ou de um processo de desenvolvimento. A figura 1.4 apresenta um exemplo do mesmo.

1.4.5 Diagrama de Sequência

O diagrama de sequência é um diagrama comportamental que preocupa-se com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em um determinado processo. Em geral, baseia-se em um caso de uso definido pelo diagrama de mesmo nome e apoia-se no diagrama de classes para determinar os objetos das classes envolvidas em um processo. Um diagrama de sequência costuma identificar o evento gerador do processo modelado, bem como o ator responsável por esse evento, e determina como o processo deve se desenrolar e ser concluído por meio da chamada de métodos disparados por mensagens enviadas entre os objetos. A figura 1.5 apresenta um exemplo desse diagrama.

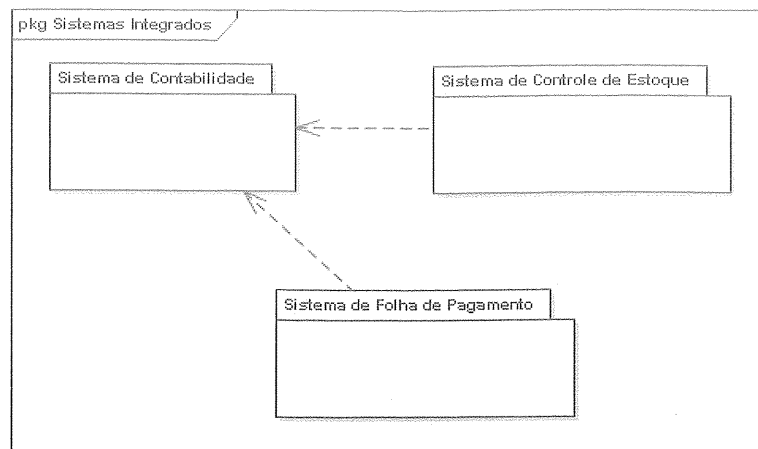


Figura 1.4 – Exemplo de Diagrama de Pacotes.

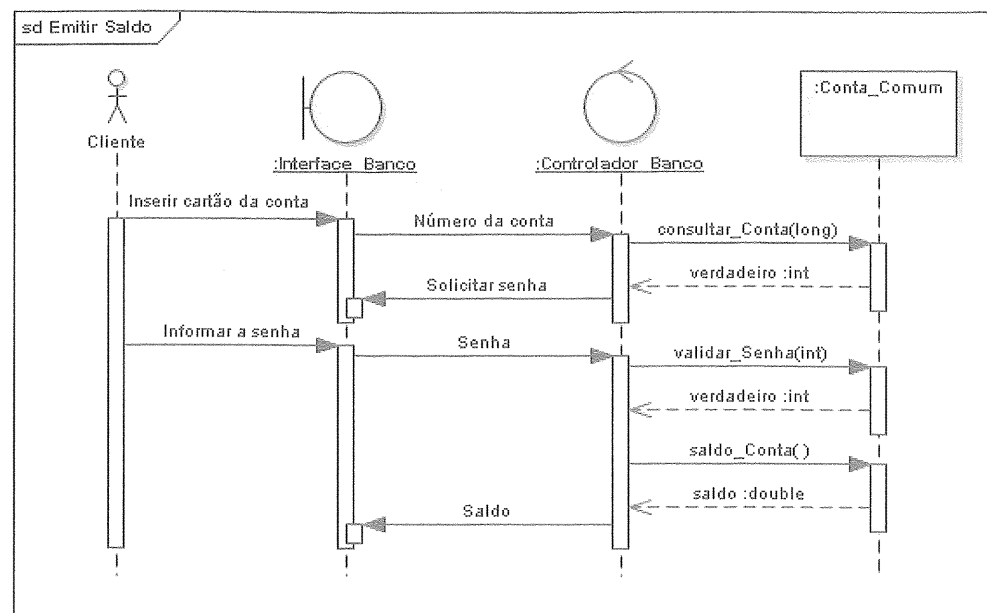


Figura 1.5 – Exemplo de Diagrama de Sequência.

1.4.6 Diagrama de Comunicação

O diagrama de comunicação era conhecido como de colaboração até a versão 1.5 da UML, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0. Está amplamente associado ao diagrama de sequência: na verdade, um complementa o outro. As informações mostradas no diagrama de comunicação com frequência são praticamente as mesmas apresentadas no de sequência, porém com um enfoque distinto, visto que esse diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante o processo. A figura 1.6 apresenta um exemplo de diagrama de comunicação.

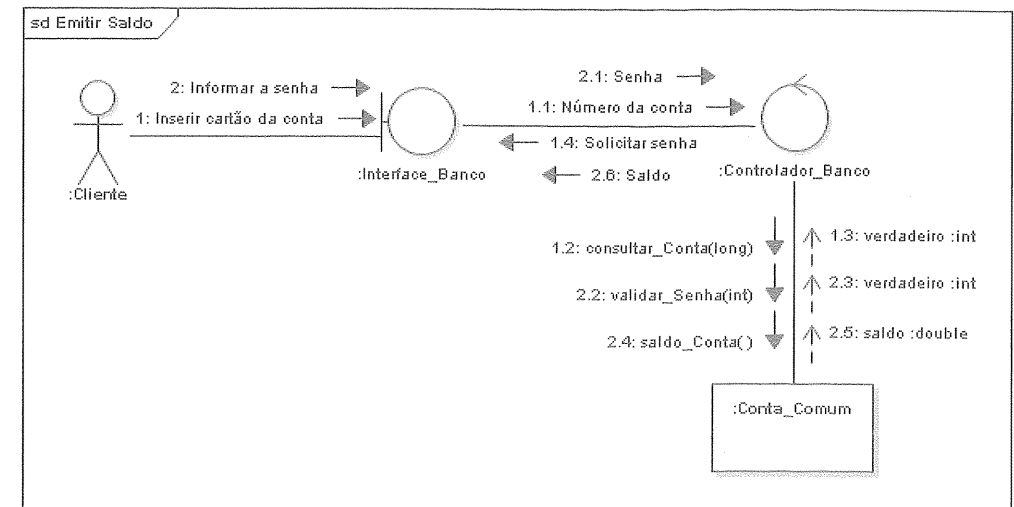


Figura 1.6 – Exemplo de Diagrama de Comunicação.

1.4.7 Diagrama de Máquina de Estados

O diagrama de máquina de estados demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado, ou seja, uma máquina de estados. Além de poder ser utilizado para expressar o comportamento de uma parte do sistema, quando é chamado de máquina de estado comportamental, também pode ser usado para expressar o protocolo de uso de parte de um sistema, quando identifica uma máquina de estado de protocolo. Como o diagrama de sequência, o de máquina de estados pode basear-se em um caso de uso, mas também pode ser utilizado para acompanhar os estados de outros elementos, como, por exemplo, uma instância de uma classe. A figura 1.7 apresenta um exemplo de diagrama de máquina de estados.

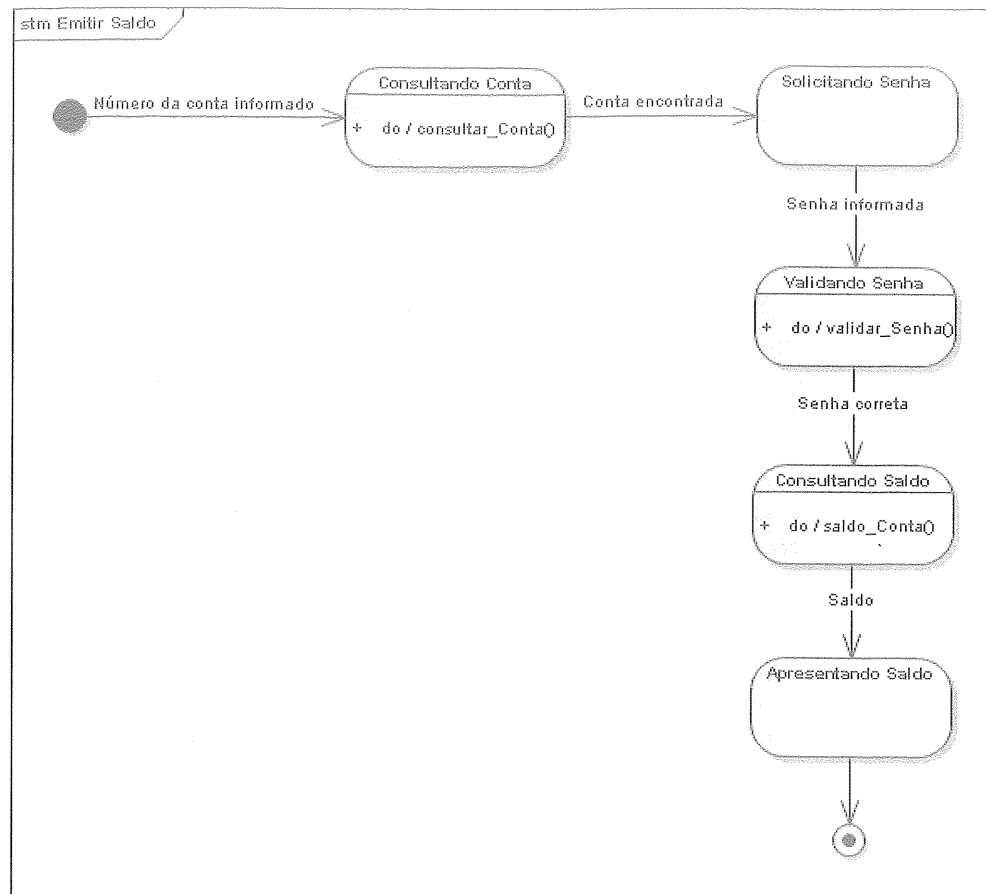


Figura 1.7 – Exemplo de Diagrama de Gráfico de Estados.

1.4.8 Diagrama de Atividade

O diagrama de atividade era considerado um caso especial do antigo diagrama de gráfico de estados, hoje conhecido como diagrama de máquina de estados, conforme descrito na seção anterior. A partir da UML 2.0, foi considerado independente do diagrama de máquina de estados. O diagrama de atividade preocupa-se em descrever os passos a serem percorridos para a conclusão de uma atividade específica, podendo esta ser representada por um método com certo grau de complexidade, um algoritmo, ou mesmo por um processo completo. O diagrama de atividade concentra-se na representação do fluxo de controle de uma atividade. A figura 1.8 apresenta um exemplo desse diagrama.

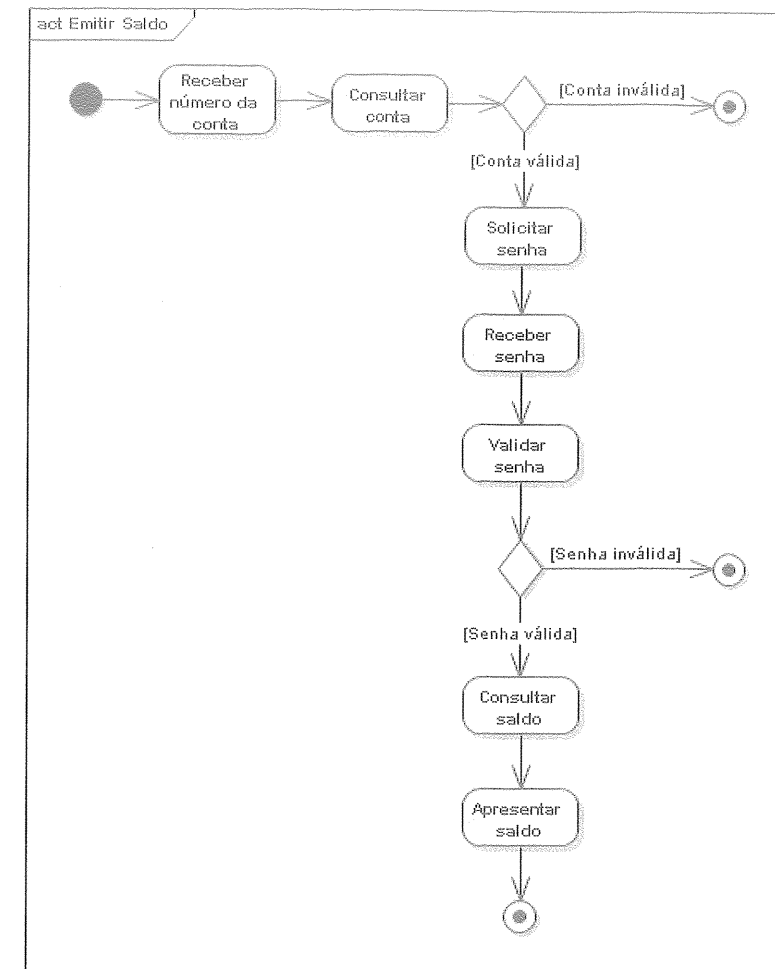


Figura 1.8 – Exemplo de Diagrama de Atividade.

1.4.9 Diagrama de Visão Geral de Interação

O diagrama de visão geral de interação é uma variação do diagrama de atividade que fornece uma visão geral dentro de um sistema ou processo de negócio. Esse diagrama passou a existir apenas a partir da UML 2. A figura 1.9 apresenta um exemplo do diagrama em questão.

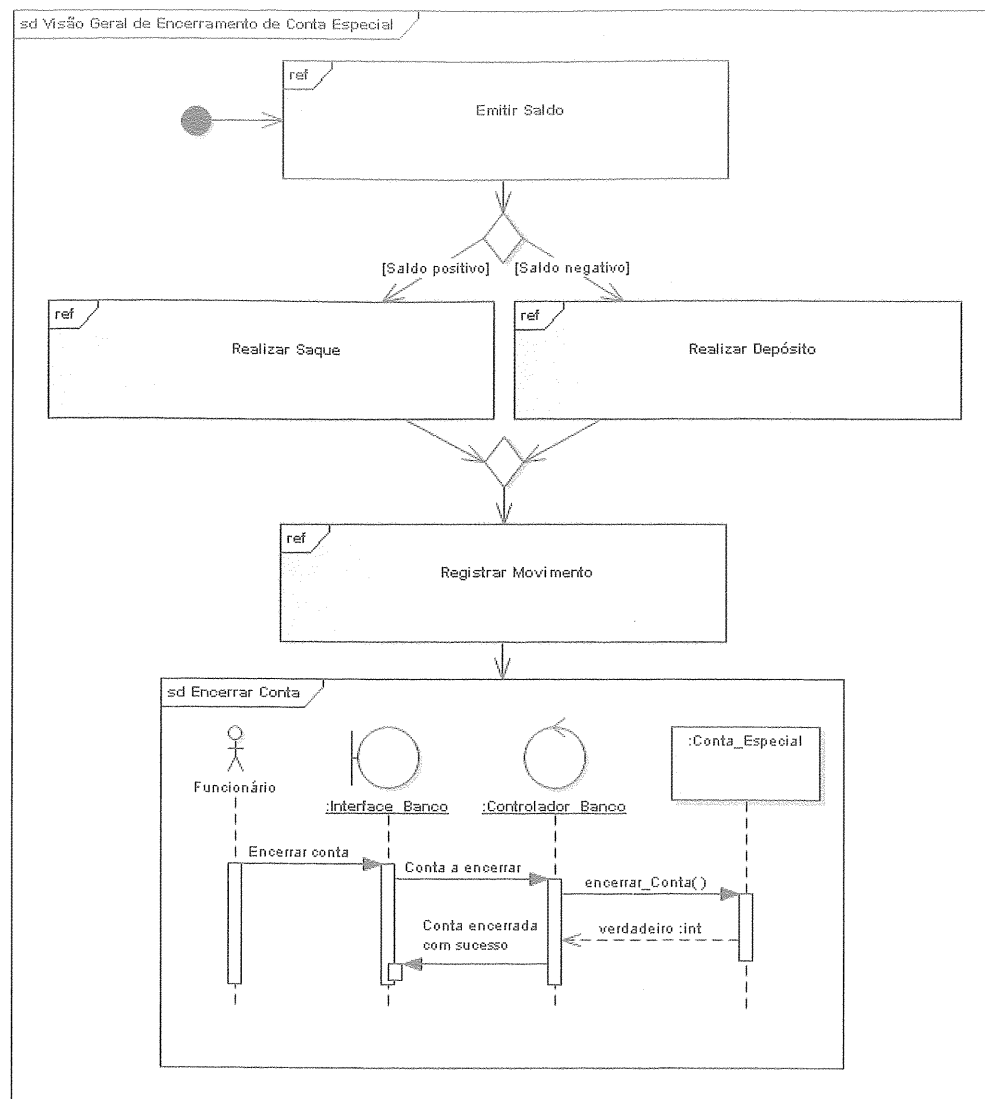


Figura 1.9 – Exemplo de Diagrama de Visão Geral de Interação.

1.4.10 Diagrama de Componentes

O diagrama de componentes está amplamente associado à linguagem de programação que será utilizada para desenvolver o sistema modelado. Esse diagrama representa os componentes do sistema quando o mesmo for ser implementado em termos de módulos de código-fonte, bibliotecas, formulários, arquivos de ajuda, módulos executáveis etc. e determina como tais componentes estarão estruturados e irão interagir para que o sistema funcione de maneira adequada. A figura 1.10 apresenta um exemplo desse diagrama.

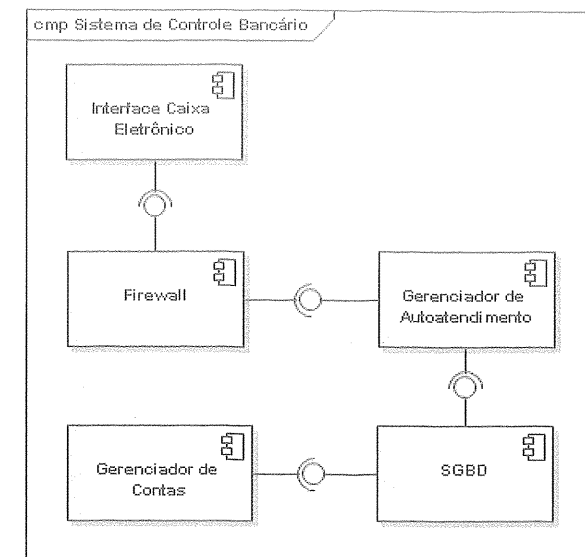


Figura 1.10 – Exemplo de Diagrama de Componentes.

1.4.11 Diagrama de Implantação

O diagrama de implantação determina as necessidades de hardware do sistema, as características físicas como servidores, estações, topologias e protocolos de comunicação, ou seja, todo o aparato físico sobre o qual o sistema deverá ser executado. Esse diagrama permite demonstrar também como se dará a distribuição dos módulos do sistema, em situações em que estes forem ser executados em mais de um servidor. A figura 1.11 apresenta um exemplo desse diagrama.

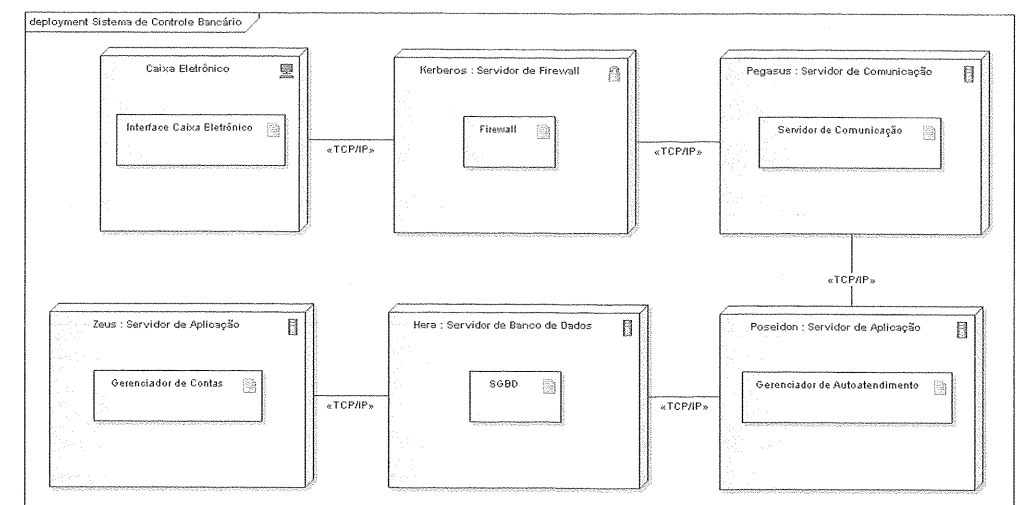


Figura 1.11 – Exemplo de Diagrama de Implantação.

1.4.12 Diagrama de Estrutura Composta

O diagrama de estrutura composta descreve a estrutura interna de um classificador, como uma classe ou componente, detalhando as partes internas que o compõem, como estas se comunicam e colaboram entre si. Também é utilizado para descrever uma colaboração em que um conjunto de instâncias cooperam entre si para realizar uma tarefa. A figura 1.12 mostra um exemplo de diagrama de estrutura composta.

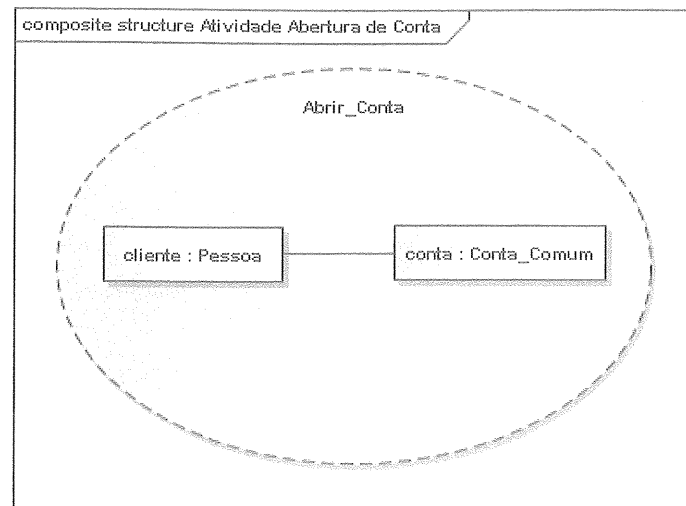


Figura 1.12 – Exemplo de Diagrama de Estrutura Composta.

1.4.13 Diagrama de Tempo ou de Temporização

O diagrama de tempo descreve a mudança no estado ou condição de uma instância de uma classe ou seu papel durante um período. Tipicamente utilizado para demonstrar a mudança no estado de um objeto no tempo em resposta a eventos externos. A figura 1.13 apresenta um exemplo desse diagrama.

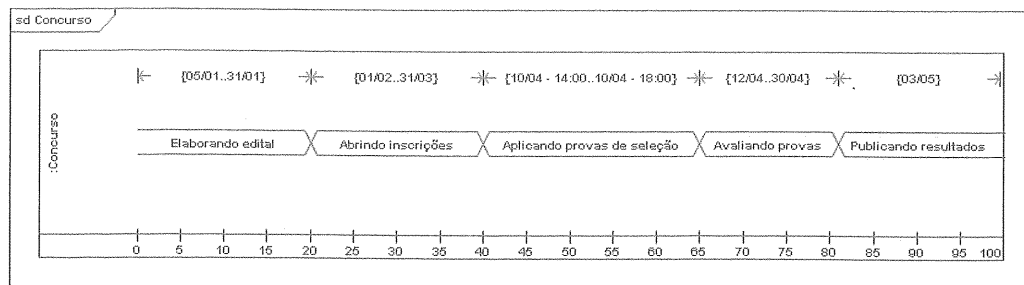


Figura 1.13 – Exemplo de Diagrama de Tempo.

1.4.14 Síntese Geral dos Diagramas

Os diagramas da UML dividem-se em diagramas estruturais e diagramas comportamentais, sendo que os últimos contêm ainda uma subdivisão representada pelos diagramas de interação, conforme pode ser verificado na figura 1.14.

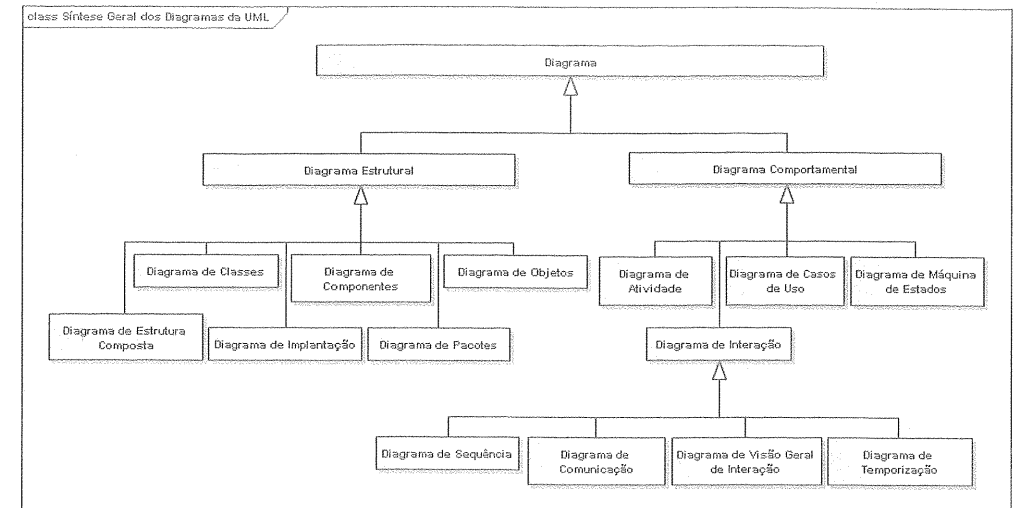


Figura 1.14 – Diagramas da UML.

Como podemos observar, os diagramas estruturais abrangem os diagramas de classes, de estrutura composta, de objetos, de componentes, de implantação e de pacotes, enquanto os comportamentais englobam os de casos de uso, atividade, máquina de estados, sequência, comunicação, visão geral de interação e tempo, sendo que os últimos quatro correspondem aos diagramas da subdivisão de diagramas de interação.

1.5 Ferramentas CASE Baseadas na Linguagem UML

Ferramentas CASE (Computer-Aided Software Engineering ou Engenharia de Software Auxiliada por Computador) são softwares que, de alguma maneira, colaboram para a execução de uma ou mais atividades realizadas durante o processo de engenharia de software. A maioria das ferramentas CASE atuais suporta a UML, sendo essa, em geral, sua principal característica. Entre as diversas ferramentas existentes hoje no mercado podemos citar:

- **Enterprise Architect** – é uma ferramenta muito fácil de utilizar. Embora não seja livre nem ofereça uma edição para a comunidade, é uma das ferramentas que mais oferecem recursos compatíveis com a UML em sua última versão. Apesar de não dispor de uma edição para a comunidade, a

Sparx Systems, a empresa que produz a Enterprise Architect, disponibiliza uma versão trial, que pode ser utilizada por cerca de 60 dias, no site www.sparxsystems.com.au. Praticamente todos os exemplos apresentados neste livro foram produzidos por meio dessa ferramenta.

- **Visual Paradigm for UML ou VP-UML** – a ferramenta pode ser encontrada no site www.visual-paradigm.com e oferece uma edição para a comunidade, ou seja, uma versão da ferramenta que pode ser baixada gratuitamente de sua página. Logicamente, a edição para a comunidade não suporta todos os serviços e opções disponíveis nas suas versões standard ou professional. No entanto, para quem deseja praticar a UML, a edição para a comunidade é uma boa alternativa, apresentando um ambiente amigável e de fácil compreensão. Além disso, a Visual-Paradigm oferece ainda uma cópia acadêmica da versão standard para instituições de ensino superior, que podem consegui-la solicitando-a na própria página da empresa. Tão logo a Visual-Paradigm comprovar a veracidade das informações fornecidas pela instituição, ela enviará uma licença de um ano para uso pelos professores e seus alunos. A licença precisa ser renovada anualmente.
- **Poseidon for UML** – esta ferramenta também tem uma edição para a comunidade, apresentando bem menos restrições que a edição para a comunidade da Visual-Paradigm, mas a interface da Poseidon é sensivelmente inferior à VP-UML, além de apresentar um desempenho um pouco inferior, embora ambas as ferramentas tenham sido desenvolvidas em Java. Uma cópia da Poseidon for UML pode ser adquirida no site www.gentleware.com.
- **ArgoUML** – trata-se de uma ferramenta um tanto limitada, e sua interface não é das mais amigáveis e intuitivas. Porém, apresenta uma característica bastante interessante e atrativa: é totalmente livre. O projeto ArgoUML constitui-se em um projeto acadêmico, no qual os códigos-fonte dessa ferramenta podem ser baixados e utilizados até mesmo para o desenvolvimento de ferramentas comerciais, como foi o caso da Poseidon for UML. Os usuários dessa ferramenta podem perceber muitas semelhanças entre as duas ferramentas, mas a Poseidon tem uma interface muito melhor e é, em geral, superior à ArgoUML. O projeto de código aberto ArgoUML exige apenas que quaisquer empresas que utilizarem seus códigos-fonte como base para uma nova ferramenta disponibilizem uma edição para a comunidade gratuitamente. Uma cópia da ArgoUML pode ser encontrada no site www.argouml.tigris.org.



CAPÍTULO 2

Orientação a Objetos

A UML está totalmente inserida no paradigma de orientação a objetos. Por esse motivo, para compreendê-la corretamente precisamos, antes, compreender o conceito de orientação a objetos.

2.1 Classificação, Abstração e Instanciação

Muitos profissionais consideram um tanto complexo o conceito do paradigma de orientação a objetos. No entanto, esse conceito é apenas diferente do enfoque procedural ao qual estão acostumados. Na realidade, o ser humano, no início de sua infância, aprende e pensa de uma maneira orientada a objetos, representando seu conhecimento por meio de abstrações e classificações (na verdade, continuamos fazendo isso mesmo quando adultos, mas desenvolvemos outras técnicas que também utilizamos em paralelo).

As crianças aprendem conceitos simples, tais como pessoa, carro e casa, por exemplo, e, ao fazerem isso, definem classes, ou seja, grupos de objetos, sendo que cada objeto é um exemplo de um determinado grupo, tendo as mesmas características e comportamentos de qualquer objeto do grupo em questão. A partir desse momento, qualquer coisa que tiver cabeça, tronco e membros torna-se uma pessoa, qualquer construção onde as pessoas possam entrar passa a ser uma casa e qualquer peça de metal com quatro rodas que se locomova de um lugar para outro transportando pessoas recebe a denominação de carro.

Na verdade, esse processo por si só já envolve um grande esforço de abstração, devido, por exemplo, aos carros apresentarem diferentes formatos, cores e estilos. Uma criança deve se sentir um pouco confusa no começo ao descobrir que o objeto amarelo e o objeto vermelho têm, ambos, a mesma classificação: carro. A partir desse momento, a criança precisa abstrair o conceito de carro para chegar à conclusão de que “carro” é um termo geral que se refere a muitos objetos. No entanto, cada um dos objetos-carro tem características semelhantes entre si,

por exemplo, todos têm quatro rodas e, no mínimo, duas portas, além de luzes de farol e de freio, bem como vidros frontais e laterais. Além disso, os objetos-carro podem realizar determinadas tarefas, sendo a principal delas transportar pessoas de um lugar para outro.

No momento em que a criança compreende esse conceito, ela percebe que “carro” é a denominação de um grupo, ou seja, ela abstraiu uma classe: a classe carro. Assim, sempre que ela perceber a presença de um objeto com as características já determinadas, ela concluirá que aquele objeto é um exemplo do grupo carro, ou seja, uma instância da classe carro.

Dessa maneira, nosso modo de aprender é, ao menos no início, orientado a objetos, ou seja, aprendemos por meio de classificações. Aprendemos a classificar praticamente tudo, criando grupos de objetos com características iguais, sendo que cada grupo de objetos é equivalente a uma classe. Sempre que precisamos compreender um conceito novo, criamos uma nova classe para esse conceito, muitas vezes derivando-a de classes mais simples (ou seja, conceitos mais simples), e determinamos que todo o objeto com as características dessa classe é um exemplo, uma instância dela. Assim, instanciamento constitui-se simplesmente em criar um exemplo de um tipo, um grupo, uma classe.

Quando instanciamos um objeto de uma classe, estamos criando um novo item do conjunto representado por essa classe, com as mesmas características e comportamentos de todos os outros objetos já instanciados. Além disso, depois de abstrair um conceito inicial, costumamos criar subdivisões dentro das classes, isto é, grupos dentro de grupos, o que já caracteriza um novo nível de abstração. Podemos criar subgrupos dentro da classe Carro, classificando-os por marca ou modelo, por exemplo, ou dentro da classe Pessoa, classificando os novos grupos por profissão, grau de parentesco ou status social.

No entanto, deve-se ter em mente que, apesar de terem os mesmos atributos, os objetos de uma classe não são exatamente iguais, pois cada objeto armazena valores diferentes em seus atributos. Por exemplo, todos os objetos da classe Carro possuem o atributo cor, mas um dos objetos pode ter uma cor azul, enquanto outro objeto pode ter uma cor verde. Da mesma forma, todos os objetos da classe Pessoa têm o atributo altura, mas o valor armazenado nesse atributo varia de pessoa para pessoa.

2.2 Classes de Objetos

Conforme foi dito, uma classe representa uma categoria, e os objetos são os membros ou exemplos dessa categoria.

Na UML, uma classe é representada por um retângulo que pode ter até três divisões. A primeira divisão armazena o nome pelo qual a classe é identificada, a segunda enuncia os possíveis atributos pertencentes à classe e a terceira lista os possíveis métodos que a classe contém. Em geral, uma classe tem atributos e métodos, mas é possível encontrar classes que contenham apenas uma dessas características ou mesmo nenhuma delas, como no caso de classes abstratas. A figura 2.1 apresenta um exemplo de classe.



Figura 2.1 – Exemplo de uma classe.

A figura 2.1 ilustra um exemplo de uma classe. Nesse caso identificamos uma classe chamada Carro. Observe que essa classe tem apenas uma divisão que contém seu nome, pois não é obrigatório representar uma classe totalmente expandida, embora seja mais comum encontrar exemplos assim. Além disso, a classe pode simplesmente não conter atributos e métodos, quando se tratar de classes abstratas. As demais divisões de uma classe serão explicadas nas seções a seguir.

2.3 Atributos ou Propriedades

Classes costumam definir atributos, também conhecidos como propriedades. Os atributos representam as características de uma classe, ou seja, as peculiaridades que costumam variar de um objeto para outro, como a altura em um objeto da classe Pessoa ou a cor em um objeto da classe Carro, e que permitem diferenciar um objeto de outro da mesma classe devido a tais variações.

Os atributos são apresentados na segunda divisão da classe e contêm, normalmente, duas informações: o nome que identifica o atributo e o tipo de dado que o atributo armazena, como, por exemplo, integer, float ou character. Essa última informação não é obrigatória, mas muitas vezes é útil e recomendável. Na realidade, não é exatamente a classe que contém os atributos, mas, sim, as instâncias, os objetos dessa classe. Não é realmente possível trabalhar com uma classe, apenas com suas instâncias. Por exemplo, a classe Pessoa não existe realmente: é uma abstração, uma forma de classificar e identificar um grupo de objetos semelhantes. Nunca poderemos trabalhar com a classe Pessoa propriamente dita, apenas com suas instâncias, como João, Pedro ou Paulo, que são nomes que identificam três objetos da classe Pessoa. Da mesma forma, ninguém pode morar na planta de uma casa. A planta exprime o conceito da classe e a partir dela constroem-se quantas casas sejam necessárias, e nessas casas reais

é que se pode morar, ou seja, a casa construída a partir da planta é um objeto, enquanto a planta da casa é uma classe.

Assim, os objetos têm os atributos relativos à classe à qual pertencem. Tais atributos são as características do objeto, como cor e número de portas, no caso de uma instância da classe carro. Todas as instâncias de uma mesma classe têm exatamente os mesmos atributos, no entanto, esses atributos podem assumir valores diversos. Assim, o atributo cor do objeto carro1 pode assumir o valor azul, enquanto no objeto carro2 o valor do atributo cor pode ser verde. A figura 2.2 demonstra um exemplo de classe com atributos.

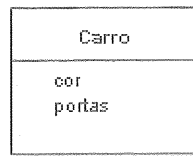


Figura 2.2 – Exemplo de classe com atributos.

O exemplo apresentado pela figura 2.2 toma a mesma classe ilustrada na figura 2.1, mas, dessa vez, com duas divisões. A segunda divisão armazena os atributos da classe carro, que nesse caso são “cor” e “portas”. No exemplo não foram definidos os tipos dos atributos, por ainda não serem necessários.

2.4 Métodos, Operações ou Comportamentos

Classes costumam ter métodos, também conhecidos como operações ou comportamentos (A UML usa o termo operação). Um método representa uma atividade que um objeto de uma classe pode executar. Para ajudar na compreensão deste conceito por profissionais não familiarizados com a orientação a objetos, podemos comparar um método a uma função como as desenvolvidas na linguagem de programação C. Da mesma forma que nas funções escritas em C, um método pode receber ou não parâmetros (valores que são utilizados durante a execução do método) e, em geral, como nas funções C, um método tende a retornar valores. Esses valores podem representar o resultado da operação executada ou simplesmente indicar se o processo foi concluído com sucesso ou não.

Assim, um método representa um conjunto de instruções que são executadas quando o método é chamado, por exemplo, um objeto da classe Carro pode executar a atividade de transportar pessoas. Grande parte da codificação propriamente dita dos sistemas de informação orientados a objeto está contida nos métodos definidos em suas classes. Os métodos são armazenados na terceira divisão de uma classe. A figura 2.3 apresenta um exemplo de uma classe com métodos.

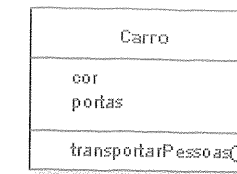


Figura 2.3 – Exemplo de classe com métodos.

Novamente tomamos a mesma classe Carro, ilustrada na figura 2.1, e acrescentamos uma terceira divisão para armazenar o método `transportarPessoas()`.

2.5 Visibilidade

A visibilidade é utilizada para indicar o nível de acessibilidade de um determinado atributo ou método, sendo representada à esquerda destes. Existem basicamente quatro modos de visibilidade: público, protegido, privado e pacote.

- A visibilidade privada é representada por um símbolo de menos (-) e significa que somente os objetos da classe detentora do atributo ou método poderão enxergá-lo ou utilizá-lo.
- A visibilidade protegida é representada pelo símbolo de sustenido (#) e determina que além dos objetos da classe detentora do atributo ou método também os objetos de suas subclasses poderão ter acesso ao mesmo.
- A visibilidade pública é representada por um símbolo de mais (+) e determina que o atributo ou método pode ser utilizado por qualquer objeto.
- A visibilidade pacote é representada por um símbolo de til (~) e determina que o atributo ou método é visível por qualquer objeto dentro do pacote. Somente elementos que fazem parte de um pacote podem ter essa visibilidade. Nenhum elemento fora do pacote poderá ter acesso a um atributo ou método com essa visibilidade.

A figura 2.4 apresenta um exemplo de classe com atributos e métodos com sua visibilidade representada à esquerda de seus nomes.

Como se pode verificar nessa figura, utilizamos novamente o mesmo exemplo de classe anterior, acrescentando visibilidade a seus atributos e métodos. No caso, é possível percebermos que os atributos “cor” e “portas” têm visibilidade privada, pois apresentam o símbolo de menos (-) à esquerda de sua descrição e, portanto, somente as instâncias da classe Carro propriamente ditas podem enxergar esses atributos. Já o método `transportarPessoas()` tem visibilidade pública, como indica o símbolo de mais (+) acrescentado à sua descrição, o que permite que objetos de outras classes visualizem o método.

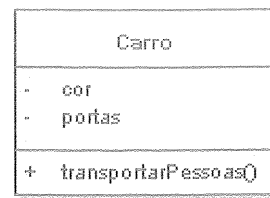


Figura 2.4 – Exemplo de Visibilidade.

2.6 Herança

Junto com o polimorfismo, que será visto a seguir, a herança é uma das características mais poderosas e importantes da orientação a objetos. Isso é devido ao fato de a herança permitir o reaproveitamento de atributos e de métodos, otimizando o tempo de desenvolvimento, além de permitir a diminuição de linhas de código, bem como facilitar futuras manutenções.

O conceito de herança trabalha com os conceitos de superclasses e subclasses. Uma superclasse, também chamada de classe-mãe, é uma classe que contém classes derivadas a partir dela, chamadas subclasses, também conhecidas como classes-filha. As subclasses, ao serem derivadas a partir de uma superclasse, herdam suas características, ou seja, seus atributos e métodos.

A vantagem do uso da herança é óbvia: ao declararmos uma classe com atributos e métodos específicos e, depois disso, derivarmos uma subclasse a partir da classe já criada, não precisamos redeclarar os atributos e métodos previamente definidos: a subclasse herda-os automaticamente, permitindo uma reutilização do código já pronto. Assim, só precisamos nos preocupar em declarar os atributos ou métodos exclusivos da subclasse, o que torna muito mais ágil o processo de desenvolvimento, além de facilitar igualmente futuras manutenções, sendo necessário apenas alterar o método da superclasse para que todas as subclasses estejam também atualizadas imediatamente.

Além disso, a herança permite trabalhar com especializações. Podemos criar classes gerais, com características compartilhadas por muitas classes, mas que tenham pequenas diferenças entre si. Assim, criamos uma classe geral com as características comuns a todas as classes e diversas subclasses a partir dela, detalhando apenas os atributos ou métodos exclusivos das mesmas. Uma subclasse pode se tornar uma superclasse a qualquer momento, bastando para tanto que se derive uma subclasse a partir dela. A figura 2.5 apresenta um exemplo de herança.

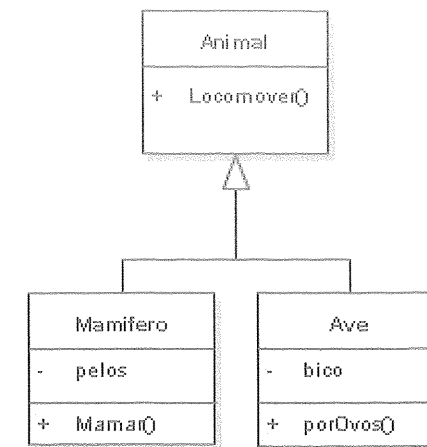


Figura 2.5 – Exemplo de Herança.

Ao observarmos a figura 2.5 percebemos a existência de uma classe geral chamada *Animal* e que essa classe tem um método chamado *Locomover()*, assim, qualquer instância da classe *Animal* pode se locomover de um lugar para outro. A partir da classe *Animal*, derivamos duas subclasses, *Mamifero* e *Ave*. A classe *Mamifero* tem como atributo a existência de pelos e como método a capacidade de mamar, enquanto a classe *Ave* tem como atributo a existência de um bico e como método a capacidade de pôr ovos, mas ambas têm a capacidade de se locomover, herdada da superclasse *Animal*.

2.6.1 Herança Múltipla

Basicamente, a herança múltipla ocorre quando uma subclasse herda características de duas ou mais superclasses. No caso, uma subclasse pode herdar atributos e métodos de diversas superclasses. No entanto, não são todas as linguagens de programação orientadas a objeto que fornecem esse tipo de recurso, sendo este encontrado, por exemplo, na linguagem C++. Veja um exemplo de herança múltipla na figura 2.6.

No exemplo da figura 2.6 aproveitamos o exemplo da figura 2.5 acrescentando uma nova subclasse derivada das classes *Mamifero* e *Ave*, a classe *Ornitorrinco*. Assim, qualquer instância da classe *Ornitorrinco* terá pelos e poderá mamar, peculiaridades herdadas da classe *Mamifero*, e também terá bico e poderá pôr ovos, peculiaridades da classe *Ave*. Observe que a visibilidade dos atributos *pelos* e *bico* é protegida, de maneira que objetos da classe *Ornitorrinco* possam enxergá-los.

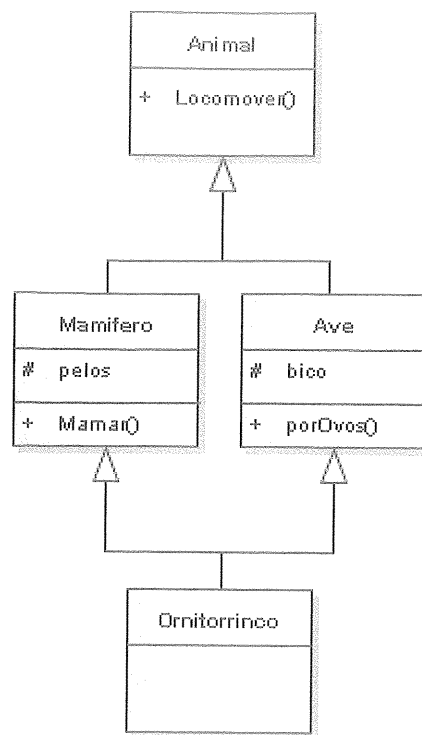


Figura 2.6 – Exemplo de Herança Múltipla.

2.7 Polimorfismo

O conceito de polimorfismo está associado à herança. O polimorfismo trabalha com a redeclaração de métodos previamente herdados por uma classe. Esses métodos, embora semelhantes, diferem de alguma forma da implementação utilizada na superclasse, sendo necessário, portanto, reimplementá-los na subclasse.

Porém, para evitar ter de modificar o código-fonte, inserindo uma chamada a um método com um nome diferente, redeclara-se o método com o mesmo nome declarado na superclasse. Assim, podem existir dois ou mais métodos com a mesma nomenclatura, diferenciando-se na maneira como foram implementados, sendo o sistema responsável por verificar se a classe da instância em questão contém o método declarado nela própria ou se o herda de uma superclasse. A figura 2.7 ilustra um exemplo de polimorfismo.

No exemplo apresentado na figura 2.7, utilizamos uma classe geral chamada Conta_Comum. Essa classe tem um atributo chamado “saldo” (com visibilidade protegida para que possa ser enxergado pelas subclasses), o qual contém o valor total depositado em uma determinada instância da classe e um método

chamado Saque. O método Saque da classe Conta_Comum simplesmente diminui o valor a ser debitado do saldo da conta e, se este não tiver o valor suficiente, a operação deverá ser recusada.

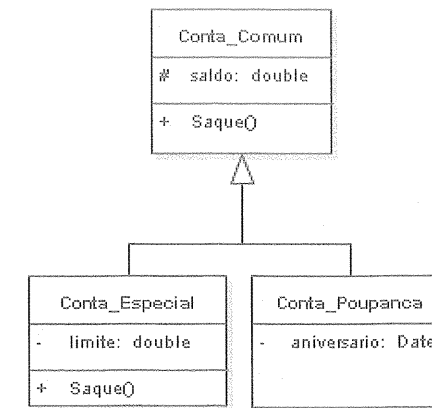


Figura 2.7 – Exemplo de Polimorfismo.

A partir da classe Conta_Comum derivamos uma nova classe chamada Conta_Especial que tem um atributo próprio, além dos herdados, chamado “limite”. Esse atributo define o valor extra que pode ser sacado além do valor contido no saldo da conta. Por esse motivo, a classe Conta_Especial apresenta uma redefinição do método Saque, porque o algoritmo do método Saque da classe Conta_Especial não é idêntico ao do método Saque declarado na classe Conta_Comum, já que é necessário incluir o limite da conta no teste para determinar se o cliente pode retirar ou não o valor solicitado. Porém, o nome do método permanece o mesmo: apenas no momento de executar o método o sistema deverá verificar se a instância que chamou o método pertence à classe Conta_Especial ou à Conta_Comum, executando o método definido na classe em questão.

O conceito de polimorfismo é um pouco semelhante ao de variáveis globais e locais utilizado pela linguagem de programação C. Ao declararmos uma variável global em um programa escrito na linguagem C, essa variável poderá ser vista e utilizada por qualquer outra função do programa. No entanto, se em alguma função declararmos uma variável local com o mesmo nome da variável global, naquela função específica, quando nos referirmos à variável em questão será a variável local, e não à global.

O mesmo ocorre com os métodos polimórficos, ou seja, supondo existirem diversas subclasses que herdam um método de uma superclasse, se uma delas redeclara esse método, ele só se comportará de maneira diferente nos objetos da classe que o modificou, permanecendo igual à forma como foi implementado na superclasse para os objetos de todas as demais classes.

CAPÍTULO 3

Diagrama de Casos de Uso

O diagrama de casos de uso procura, por meio de uma linguagem simples, possibilitar a compreensão do comportamento externo do sistema (em termos de funcionalidades oferecidas por ele) por qualquer pessoa, tentando apresentar o sistema por intermédio de uma perspectiva do usuário. É, entre todos os diagramas da UML, o mais abstrato e, portanto, o mais flexível e informal. Esse diagrama costuma ser utilizado, sobretudo no início da modelagem do sistema, principalmente nas etapas de levantamento e análise de requisitos, embora venha a ser consultado e possivelmente modificado durante todo o processo de engenharia e sirva de base para a modelagem de outros diagramas.

Esse diagrama tem por objetivo apresentar uma visão externa geral das funcionalidades que o sistema deverá oferecer aos usuários, sem se preocupar com a questão de como tais funcionalidades serão implementadas. O diagrama de casos de uso é de grande auxílio para a identificação e compreensão dos requisitos do sistema, ajudando a especificar, visualizar e documentar as características, funções e serviços do sistema desejados pelo usuário. O diagrama de casos de uso tenta identificar os tipos de usuários que irão interagir com o sistema, quais papéis esses usuários irão assumir e quais funções um usuário específico poderá requisitar.

Por utilizar uma linguagem informal e apresentar uma visão geral do comportamento do sistema a ser desenvolvido, o diagrama de casos de uso pode e deve ser apresentado durante as reuniões iniciais com os clientes como uma forma de ilustrar o comportamento do sistema de informação, facilitar a compreensão dos usuários e auxiliar na identificação de possíveis falhas de especificação, verificando se os requisitos do sistema foram bem compreendidos. É bastante útil e recomendável que um diagrama de casos de uso seja apresentado aos clientes junto com um protótipo, o que permitirá que um complemente o outro.

3.1 Atores

O diagrama de casos de uso concentra-se em dois itens principais: atores e casos de uso. Os atores representam os papéis desempenhados pelos diversos usuários que poderão utilizar, de alguma maneira, os serviços e funções do sistema. Eventualmente um ator pode representar algum hardware especial ou mesmo outro software que interaja com o sistema, como no caso de um sistema integrado, por exemplo. Assim, um ator pode ser qualquer elemento externo que interaja com o software. Os atores são representados por símbolos de “bonecos magros”, contendo uma breve descrição logo abaixo de seu símbolo que identifica o papel que o ator em questão assume dentro do diagrama. A figura 3.1 apresenta alguns exemplos de atores.

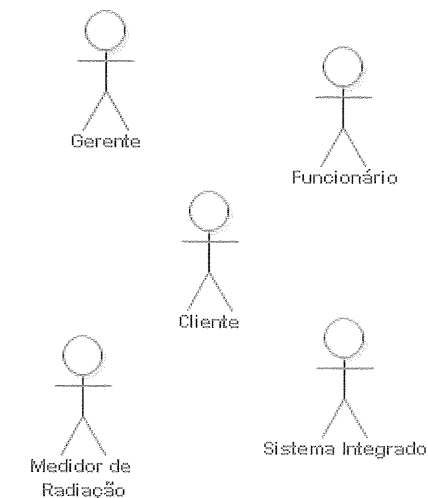


Figura 3.1 – Exemplos de Atores.

Nesse exemplo, os atores **Gerente**, **Funcionário** e **Cliente** representam usuários normais, enquanto o ator **Medidor de Radiação** representa um hardware externo que envia informações para o sistema. Já o ator **Sistema Integrado** representa um software que interage de alguma forma com o sistema.

3.2 Casos de Uso

Os casos de uso são utilizados para capturar os requisitos do sistema, ou seja, referem-se aos serviços, tarefas ou funcionalidades identificados como necessários ao software e que podem ser utilizados de alguma maneira pelos atores que interagem com o sistema, sendo usados para expressar e documentar os

comportamentos pretendidos para as funções deste. Podem ser classificados em casos de uso primários ou secundários. Um caso de uso é considerado primário quando se refere a um processo importante, que enfoca um dos requisitos funcionais do software, como realizar um saque ou emitir um extrato em um sistema de controle bancário. Já um caso de uso secundário se refere a um processo periférico, como a manutenção de um cadastro.

Pode-se, às vezes, associar um caso de uso a um formulário do sistema, embora isso não seja de forma alguma uma regra, já que um caso de uso, ou seja, uma funcionalidade do sistema, pode perfeitamente abranger vários formulários do sistema ou constituir-se em apenas uma pequena opção de uma interface, como um botão, por exemplo.

Os casos de uso são representados por elipses contendo dentro de si um texto que descreve a que funcionalidade o caso de uso se refere. Na verdade, não existe um limite determinado para o texto que possa estar contido dentro da elipse, mas em geral, a descrição de um caso de uso costuma ser bastante sucinta. A figura 3.2 apresenta um exemplo de caso de uso, representando a abertura de uma conta corrente.



Figura 3.2 – Exemplo de Caso de Uso.

Os casos de uso costumam ser documentados, fornecendo instruções em linhas gerais de como será seu funcionamento, quais atividades deverão ser executadas, qual evento forçará sua execução, quais atores poderão utilizá-los e quais suas possíveis restrições, entre outras.

Embora a documentação seja utilizada no momento da implementação de um caso de uso, além de servir como base para a construção de outros diagramas baseados nos casos de uso, como é comum acontecer com o diagrama de sequência, que será visto nos capítulos seguintes, o objetivo principal da documentação de um caso de uso é fornecer um relatório ao cliente explicando qual o comportamento pretendido para um determinado caso de uso e quais funções ele executará quando for solicitado. Normalmente, um caso de uso é documentado de maneira informal, mas nada impede que o engenheiro de software insira detalhes de implementação em uma linguagem mais técnica, se assim considerar necessário.

3.3 Documentação de Casos de Uso

A documentação de um caso de uso costuma descrever, por meio de uma linguagem bastante simples, informações como a função em linhas gerais do caso de uso, quais atores interagem com ele, quais etapas devem ser executadas pelo ator e pelo sistema para que o caso de uso execute sua função, quais parâmetros devem ser fornecidos e quais restrições e validações o caso de uso deve ter.

No entanto, não existe um formato específico de documentação para casos de uso definidos pela UML, o que está de acordo com a característica do próprio diagrama, ou seja, o formato de documentação de um caso de uso é bastante flexível, permitindo que se documente o caso de uso da forma que se considerar melhor, até mesmo por meio do uso de pseudocódigo ou da utilização do código de uma linguagem de programação propriamente dita, embora isso fuja bastante do objetivo principal do diagrama, que é utilizar uma linguagem simples que até mesmo usuários leigos possam entender.

Por esse motivo, recomendamos que se evite a utilização de pseudocódigo na documentação de casos de uso, pois isso é mais adequado em outros diagramas, como o de atividade, por exemplo. Os casos de uso podem também ser documentados por meio de outros diagramas, como o de sequência, de máquina de estados ou de atividade. A tabela 3.1 apresenta uma sugestão de como documentar o caso de uso representado na figura 3.2.

Tabela 3.1 – Documentação do Caso de Uso Abertura de Conta

Nome do Caso de Uso	Abrir Conta
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	Funcionário
Resumo	Esse caso de uso descreve as etapas percorridas por um cliente para abrir uma conta corrente
Pré-Condições	O pedido de abertura precisa ter sido previamente aprovado
Pós-Condições	É necessário realizar um depósito inicial
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Solicitar Abertura de Conta	
	2. Consultar cliente por seu CPF ou CNPJ
3. Informar a senha da conta	
	4. Abrir conta
5. Fornecer valor a ser depositado	
	6. Registrar depósito
	7. Emitir cartão da conta

Restrições/Validações	1. Para abrir uma conta corrente é preciso ser maior de idade
	2. O valor mínimo de depósito é R\$ 5,00
	3. O cliente precisa fornecer algum comprovante de residência
Fluxo Alternativo – Manutenção do Cadastro do Cliente	
Ações do Ator	Ações do Sistema
	1. Se for necessário, Executar Caso de Uso Manter Cliente, para gravar ou atualizar o cadastro do cliente.
Fluxo de Exceção – Cliente menor de idade	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que este não possui a idade mínima para possuir uma conta corrente
	2. Recusar o pedido

Seguindo o modelo apresentado na tabela 3.1, em primeiro lugar deve-se fornecer uma descrição para o caso de uso que está sendo documentado, em geral a mesma descrição apresentada externamente no caso de uso.

A informação seguinte talvez aparente ser um pouco mais complexa: o que significa “Caso de Uso Geral”? Como será visto nas próximas seções, pode haver casos de uso gerais e casos de uso especializados, que herdaram as características dos casos de uso gerais. Assim, no item caso de uso geral, deve-se informar, se existir, o nome do caso de uso geral a partir do qual o caso de uso atual foi derivado. Nesse exemplo, não existe um caso de uso geral e, por isso, o campo foi deixado em branco. Porém, se o caso de uso em questão representasse a abertura de uma conta especial derivada a partir do caso de uso “Abrir Conta Comum”, o campo caso de uso geral armazenaria o texto “Abrir Conta Comum”, indicando que o caso de uso documentado é uma especialização do caso de uso “Abrir Conta Comum”.

O campo ator principal identifica o ator que mais interage com o caso de uso ou que está mais interessado nos resultados produzidos pelo mesmo. Já os atores secundários são aqueles que interagem em um nível menor com o caso de uso ou que não têm tanto interesse em seus resultados. Nesse exemplo foram identificados dois atores, **Cliente** e **Funcionário**. O ator principal é o **Cliente**, porque é ele quem solicita o serviço, embora não interaja diretamente com o sistema, é quem está mais interessado no resultado do processo e quem fornece a maioria das informações. Porém, quem interage realmente com o serviço e manipula a funcionalidade de abertura de conta é o ator **Funcionário**, uma pessoa contratada pela instituição bancária para atender seus clientes. Na documentação desse caso de uso, as ações tomadas pelo ator **Funcionário** são representadas como ações do sistema.

O campo seguinte da documentação desse caso de uso apresenta um breve resumo explicando seu objetivo. Em seguida, são descritas as possíveis pré-condições para que o caso de uso seja executado ou concluído e as possíveis pós-condições, ou seja, tarefas que devem ser realizadas depois que as etapas do caso de uso tiverem sido concluídas. Nesse exemplo, para que a abertura de conta seja realizada, é preciso primeiro, como pré-condição, que o pedido de abertura do cliente seja aprovado e, após a abertura da conta, é necessário depositar algum valor na mesma, o que caracteriza uma pós-condição.

Em seguida, passa-se ao fluxo principal do caso de uso, que apresenta as ações que devem normalmente ser realizadas quando o serviço representado pelo caso de uso for solicitado. As ações são divididas em ações realizadas pelo ator que interage com o sistema e em ações realizadas pelo próprio sistema, numeradas em uma ordem sequencial. Pode haver casos em que existam fluxos alternativos que representam situações que fogem da situação ideal do caso de uso ou que representem opções que podem ser executadas ou não, dependendo se uma condição for satisfeita ou não. Nesse exemplo, existe um fluxo alternativo que representa uma situação em que o cliente não possui cadastro ou este precisa ser atualizado, havendo a necessidade de executar os passos de outro caso de uso para realizar essa manutenção. Pode existir ainda fluxos de exceção que determinam as ações que devem ser tomadas em situações em que o fluxo não possa ser concluído, devido a alguma regra de negócio ter sido quebrada, por exemplo. Nesse exemplo há um fluxo de exceção que trata uma situação em que a regra que determina que para abrir uma conta é necessário ser maior de idade foi quebrada.

Finalmente, é apresentada uma lista de possíveis restrições e validações do caso de uso, com o objetivo de tornar o processo mais consistente. Na documentação desse caso de uso determinou-se que o cliente precisa ser maior de idade para abrir uma conta corrente, o que constitui uma restrição. Além disso, fixou-se o valor mínimo para depósito em R\$ 5,00, e isso precisa ser validado no processo de abertura de conta, ou seja, o sistema não pode aceitar um depósito com um valor inferior a esse no momento em que uma conta for aberta. Essas restrições caracterizam as regras do negócio da empresa, ou seja, regras que determinam as condições para que o processo seja executado.

3.4 Associações

As associações representam as interações ou relacionamentos entre os atores que fazem parte do diagrama, entre os atores e os casos de uso ou os relacionamentos entre os casos de uso e outros casos de uso. Os relacionamentos entre casos de

uso recebem nomes especiais, como inclusão, extensão e generalização. Todos esses relacionamentos serão examinados no decorrer das próximas seções.

Uma associação entre um ator e um caso de uso demonstra que o ator utiliza, de alguma maneira, a funcionalidade do sistema representada pelo caso de uso em questão, seja requisitando a execução daquela função, seja recebendo o resultado produzido por ela a pedido de outro ator.

A associação entre um ator e um caso de uso é representada por uma linha ligando o ator ao caso de uso, podendo ocorrer que as extremidades da linha contenham setas, indicando o sentido em que as informações trafegam, ou seja, se estas são fornecidas pelo ator ao caso de uso, se são transmitidas pelo caso de uso ao ator ou ambos (nesse último caso, a linha não tem setas, significando que as informações são transmitidas nas duas direções). As setas também servem para indicar quem inicia a comunicação. Além disso, uma associação pode ter uma descrição própria quando há necessidade de esclarecer a natureza da informação que está sendo transmitida ou para dar um nome à associação, se isso for necessário. A figura 3.3 representa uma associação entre um ator e um caso de uso.

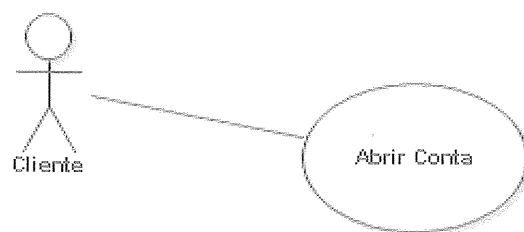


Figura 3.3 – Associação entre um Ator e um Caso de Uso.

Ao examinarmos o exemplo ilustrado pela figura 3.3, percebemos que o ator **Cliente** utiliza, de alguma forma, a funcionalidade de **Abrir Conta**, representada pelo caso de uso, e que a informação referente a esse processo trafega nas duas direções.

3.5 Generalização/Especialização

O relacionamento de generalização/especialização é uma forma de associação entre casos de uso na qual existem dois ou mais casos de uso com características semelhantes, apresentando pequenas diferenças entre si. Quando tal situação ocorre, costuma-se definir um caso de uso geral que descreve as características compartilhadas por todos os casos de uso em questão e então relacioná-lo com os outros casos de uso envolvidos, cuja documentação conterá apenas as características específicas de cada um.

Assim, é desnecessário colocar a mesma documentação para todos os casos de uso envolvidos, pois toda a estrutura de um caso de uso generalizado é herdada pelos casos de uso especializados. Além disso, os casos de uso especializados herdam também quaisquer possíveis associações de inclusão ou extensão que o caso de uso geral venha a ter, bem como quaisquer associações com os atores que utilizem o caso de uso geral. A associação de generalização/especialização é representada por uma linha com uma seta mais grossa, que indica qual o caso de uso geral (para o qual a seta aponta) e quais os casos de uso especializados (os que se encontram na outra extremidade da seta, apontando para o caso de uso geral). Um exemplo de generalização/especialização pode ser visto na figura 3.4.

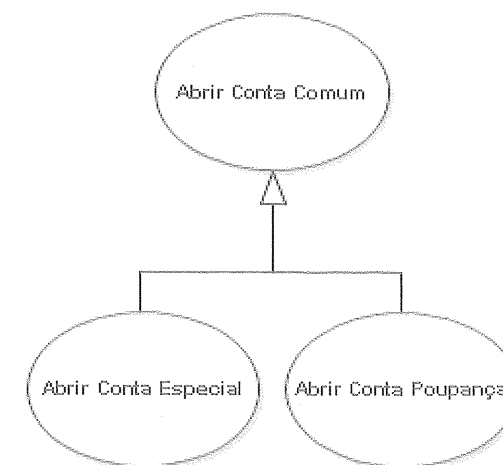


Figura 3.4 – Generalização/Especialização.

Como se pode perceber no exemplo da figura 3.4, há três opções de abertura de conta muito semelhantes entre si: abertura de conta comum, abertura de conta especial e abertura de conta-poupança, cada uma representada por um caso de uso diferente. Os processos de abertura de conta especial e de conta-poupança são muito semelhantes ao de abertura de conta comum, mas têm algumas características próprias, o que justifica colocá-las como especializações do caso de uso **Abrir Conta Comum**, detalhando-se as particularidades de cada caso de uso especializado em sua própria documentação.

Nesse exemplo, o caso de uso **Abrir Conta Especial** deve incluir a definição do limite da conta no momento de sua aprovação, enquanto o caso de uso **Abrir Conta-Poupança** não apresenta a maioria das restrições e validações necessárias à abertura de uma conta corrente, como a obrigatoriedade de o cliente ser maior de idade, por exemplo.

O relacionamento de generalização/especialização também pode ser aplicado sobre atores. A figura 3.5 apresenta um exemplo de generalização/especialização entre atores em que existe um ator geral chamado *Pessoa* e dois atores especializados chamados respectivamente *Pessoa Física* e *Pessoa Jurídica*.

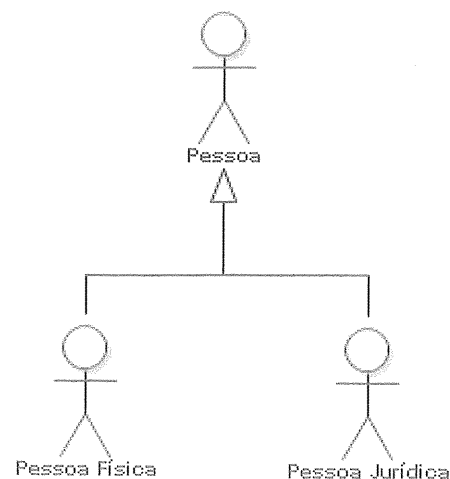


Figura 3.5 – Generalização/ Especialização com Atores.

Um outro exemplo desse tipo de relacionamento aplicado a atores pode ser visto na figura 3.6, na qual temos três atores, *Usuário Júnior*, *Usuário Sênior* e *Administrador*. Nesse exemplo, o ator *Usuário Júnior* pode apenas executar o caso de uso *Ler Arquivo*, já o ator *Usuário Sênior* pode executar o caso de uso *Ler Arquivo*, por ser uma especialização do ator *Usuário Júnior*, e executar o caso de uso *Gravar Arquivo*. Observe que não existe uma associação entre o ator *Usuário Sênior* e o caso de uso *Ler Arquivo*, já que isto não é necessário uma vez que o ator *Usuário Sênior* herda essa associação do ator *Usuário Júnior*. Finalmente o ator *Administrador* pode executar os casos de uso *Ler Arquivo* e *Gravar Arquivo* herdados do ator *Usuário Sênior*, além do caso de uso *Excluir Arquivo*, que somente ele pode executar. Assim, percebe-se que o relacionamento de generalização/especialização aplicado a atores pode permitir representar níveis de usuários e aproveitar associações já definidas, evitando deixar o diagrama com muitas linhas entrecruzadas.

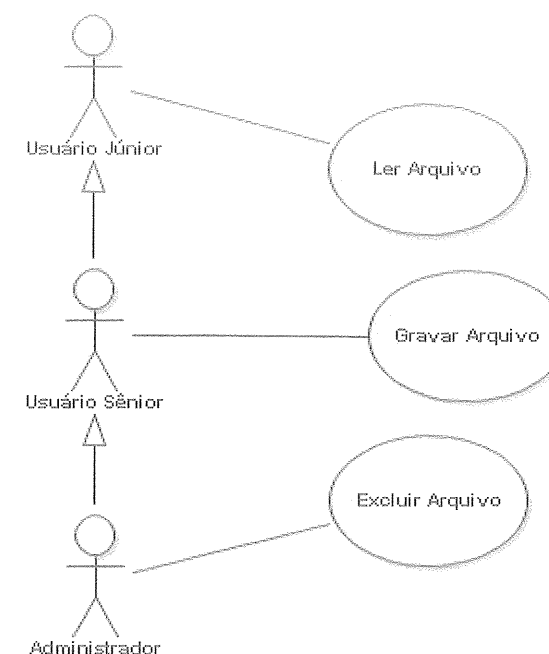


Figura 3.6 – Generalização/ Especialização com Atores e Casos de Uso.

3.6 Inclusão

A associação de inclusão costuma ser utilizada quando existe um cenário, situação ou rotina comum a mais de um caso de uso. Quando isso ocorre, a documentação dessa rotina é colocada em um caso de uso específico para que outros casos de uso utilizem esse serviço, evitando-se descrever uma mesma sequência de passos em vários casos de uso. Os relacionamentos de inclusão indicam uma obrigatoriedade, ou seja, quando um determinado caso de uso tem um relacionamento de inclusão com outro, a execução do primeiro obriga também a execução do segundo. Um relacionamento de inclusão pode ser comparado à chamada de uma sub-rotina ou função, artifício bastante utilizado na maioria das linguagens de programação.

Uma associação de inclusão é representada por uma linha tracejada contendo uma seta em uma de suas extremidades, a qual aponta para o caso de uso incluído no caso de uso posicionado na outra extremidade da linha. As associações de inclusão costumam apresentar também um estereótipo que contém o texto “include”, entre dois sinais de menor (<) e dois de maior (>). Um estereótipo serve para destacar um componente ou associação atribuindo-lhe características

especiais em relação a seus iguais. Os estereótipos serão melhor detalhados ao longo do livro. Um exemplo de inclusão pode ser examinado na figura 3.7.

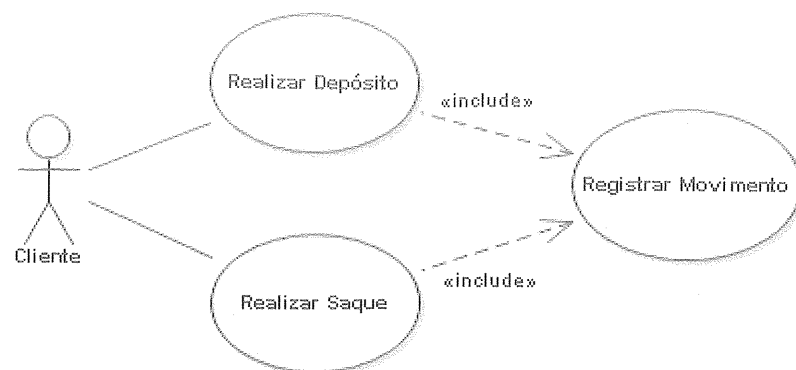


Figura 3.7 – Inclusão.

Ao analisarmos a figura 3.7, percebemos que sempre que um saque ou depósito ocorrer deve ser registrado para fins de histórico bancário. Como as rotinas para registro de um saque ou depósito são extremamente semelhantes, diferenciando-se apenas pelo tipo de movimento, colocou-se a rotina de registro em um caso de uso à parte, chamado **Registrar Movimento**. O caso de uso **Registrar Movimento** será executado obrigatoriamente sempre que os casos de uso **Realizar Depósito** ou **Realizar Saque** forem utilizados. Assim, não é preciso descrever os passos para registrar um movimento nesses casos de uso, pois as etapas estão descritas no caso de uso **Registrar Movimento**.

Outro exemplo de inclusão pode ser visto na figura 3.8, onde modelamos parte de um sistema de livreria virtual, onde o cliente pode logar-se, adicionar livros ao carrinho de compras, visualizar o conteúdo do carrinho e concluir o pedido.

Nesse exemplo, o cliente pode solicitar a visualização de seu carrinho de compras sempre que quiser. No entanto, no momento em que decidir concluir o pedido, obrigatoriamente deverá verificar ainda uma vez os livros por ele escolhidos e fornecer ainda uma última confirmação. Dessa forma, como o cliente pode executar o processo de visualização de carrinho sempre que quiser, este caracteriza-se como um processo independente da conclusão de pedido. Entretanto, como é necessário visualizar o conteúdo do carrinho antes de concluir o pedido, o caso de uso **Concluir Pedido** deverá incluir o caso de uso **Visualizar Carrinho** para não ter que repetir os passos já definidos nesse caso de uso, poupando tempo e facilitando futuras manutenções.

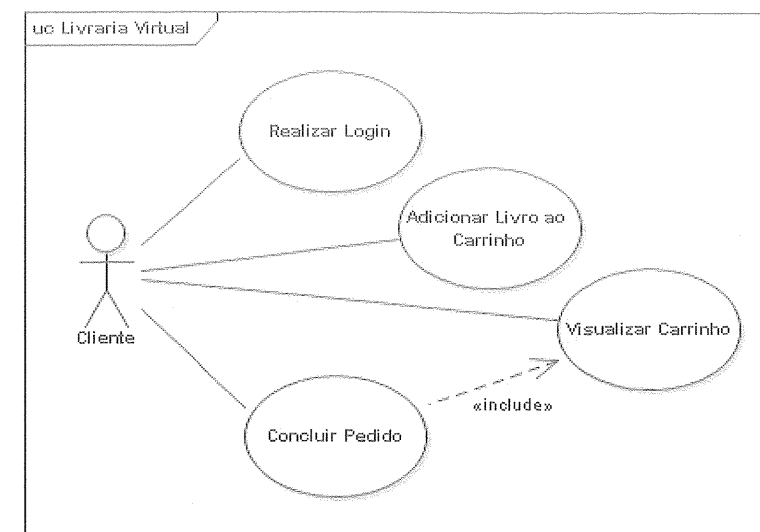


Figura 3.8 – Inclusão.

3.7 Extensão

Associações de extensão são utilizadas para descrever cenários opcionais de um caso de uso. Os casos de uso estendidos descrevem cenários que apenas ocorrerão em uma situação específica se determinada condição for satisfeita. Assim, as associações de extensão indicam a necessidade de um teste para determinar se é necessário executar também o caso de uso estendido ou não. Relacionamentos de extensão representam eventos que não ocorrem sempre, o que não significa que eles sejam incomuns. As associações de extensão têm uma representação muito semelhante às associações de inclusão, sendo também representadas por uma linha tracejada, diferenciando-se pelo fato de a seta apontar para o caso de uso que utiliza o caso de uso estendido e por haver um estereótipo contendo o texto “extend” em vez de “include”. Para ajudar a compreender o conceito de extensão, vamos ilustrar um exemplo bastante simples com um formulário apresentado na figura 3.9.

Figura 3.9 – Formulário de Login.

A figura 3.9 enfoca uma situação pela qual a maioria dos leitores provavelmente já passou, representando um formulário de login, onde o cliente deverá informar seu nome-login e senha para poder se autenticar no sistema. No fluxo ideal desse processo, o cliente informará um nome-login e uma senha válidos e lhe será permitido o acesso ao software. Contudo, pode acontecer de o cliente estar tendo acesso a esse formulário pela primeira vez e não possuir cadastro no sistema. Prevendo essa possibilidade, o formulário permite que o cliente se registre, apresentando-lhe a opção de pressionar o botão “Autorregistrar” para se cadastrar no sistema. Obviamente o cliente só fará isso na primeira vez, seguindo o processo normal nas vezes seguintes.

Uma vez que o processo de **Autorregistrar** poderá ser chamado a partir do processo de **Login**, mediante a condição de o cliente não estar cadastrado, podemos representar esse relacionamento por meio de uma associação de extensão, conforme demonstra a figura 3.10.

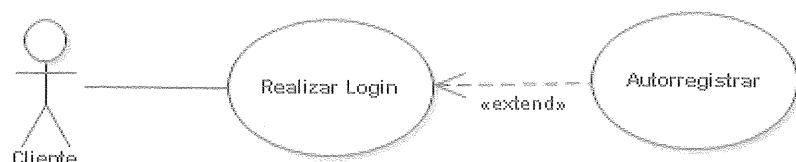


Figura 3.10 – Extensão.

Como podemos perceber, os processos de **Login** e **Autorregistrar** são representados como casos de uso e há uma associação de extensão entre eles, demonstrando que o caso de uso **Autorregistrar** poderá ser chamado a partir do caso de uso **Realizar Login**.

Um caso de uso pode ter muitos relacionamentos de extensão, conforme pode ser observado na figura 3.11, onde apresentamos um exemplo de extensão um pouco mais complexo, referente ao processo de encerramento de conta do sistema de controle bancário que temos modelado.

Nesse exemplo, um cliente dirige-se a um funcionário do banco e solicita o encerramento de uma determinada conta. Como sabemos, para encerrar uma conta é necessário que seu saldo seja igual a zero. O caso de uso **Encerrar Conta** representa a funcionalidade de encerramento de conta utilizada pelo funcionário do banco e pode, eventualmente, fazer uma chamada ao caso de uso **Realizar Saque**, se o saldo da conta estiver positivo, ou ao caso de uso **Realizar Depósito**, se o saldo da conta estiver negativo. Essas associações entre o caso de uso **Encerrar Conta** e os casos de uso **Realizar Saque** e **Realizar Depósito** são associações de extensão porque os casos de uso somente serão chamados pelo caso de uso **Encerrar Conta** se as

condições a eles associadas forem verdadeiras e, nessa situação, mesmo quando forem chamados, será somente um deles.

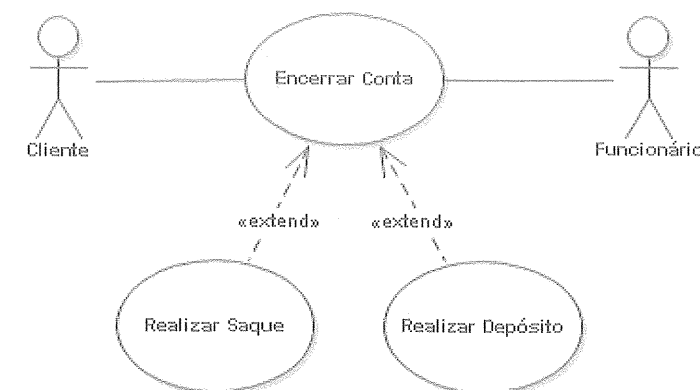


Figura 3.11 – Extensão.

3.8 Restrições em Associações de Extensão

Restrições são compostas por um texto entre chaves e utilizadas para definir validações, consistências, condições etc., que devem ser aplicadas a um determinado componente ou situação. Em se tratando de extensões, às vezes nem sempre fica claro qual é a condição para que um caso de uso estendido seja executado, assim, pode-se acrescentar uma restrição à associação de extensão por meio de uma nota explicativa, determinando a condição para que o caso de uso seja executado, conforme apresentado nas figuras 3.12 e 3.13.

No exemplo da figura 3.12 acrescentou-se uma restrição à associação de extensão para determinar a condição que necessita ser satisfeita para que o caso de uso **Autorregistrar** seja executado, ou seja, se o cliente ainda não estiver registrado.

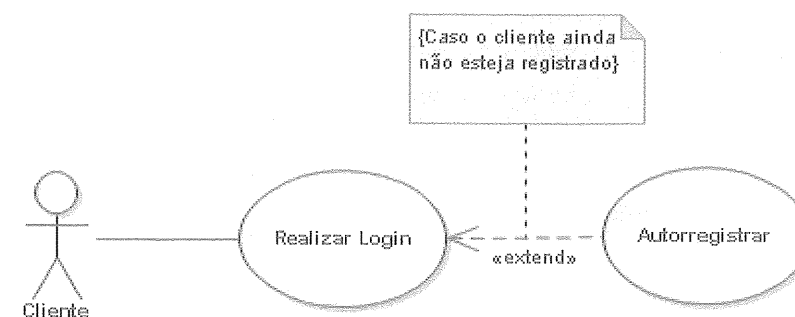


Figura 3.12 – Associação de Extensão com Restrição – Realizar Login.

Já no exemplo da figura 3.13, o caso de uso Realizar Saque só será executado se o saldo da conta a ser encerrada for positivo, e o caso de uso Realizar Depósito somente será executado quando o saldo da conta for negativo. Essas restrições nada mais são do que regras de negócio referentes ao processo de encerramento de conta. Aqui empregamos o componente nota explicativa, que é utilizado para incluir comentários ou restrições sobre um componente ou relacionamento. A seta tracejada que o une ao componente é chamada âncora.

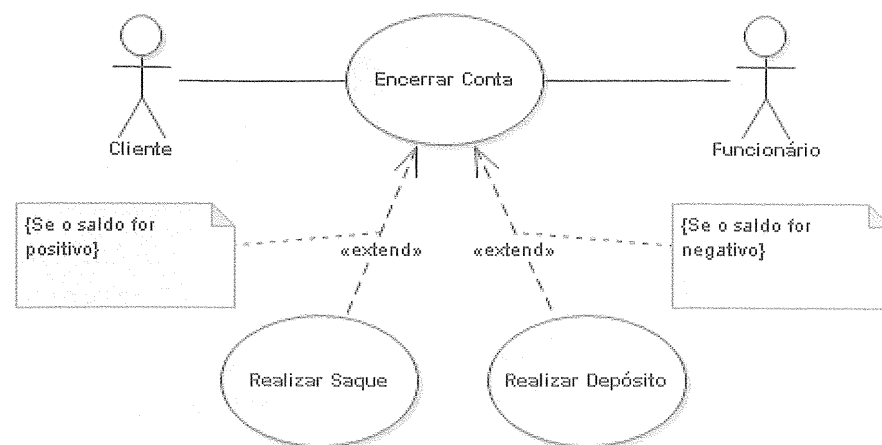


Figura 3.13 – Associação de Extensão com Restrição – Encerrar Conta.

3.9 Pontos de Extensão

Um ponto de extensão identifica um ponto no comportamento de um caso de uso a partir do qual esse comportamento poderá ser estendido pelo comportamento de outro caso de uso, se a condição para que isso ocorra for satisfeita, conforme mostra a figura 3.14.

Aqui modificamos o exemplo apresentado na figura 3.13, inserindo os pontos de extensão Saldo positivo e Saldo negativo, a partir dos quais os casos de uso Realizar Saque e Realizar Depósito poderão ser estendidos. As associações de extensão deverão estar de acordo com a documentação do caso de uso, conforme demonstra a tabela 3.2, onde os fluxos alternativos Saldo positivo e Saldo negativo fazem referência aos casos de uso Realizar Saque e Realizar Depósito.

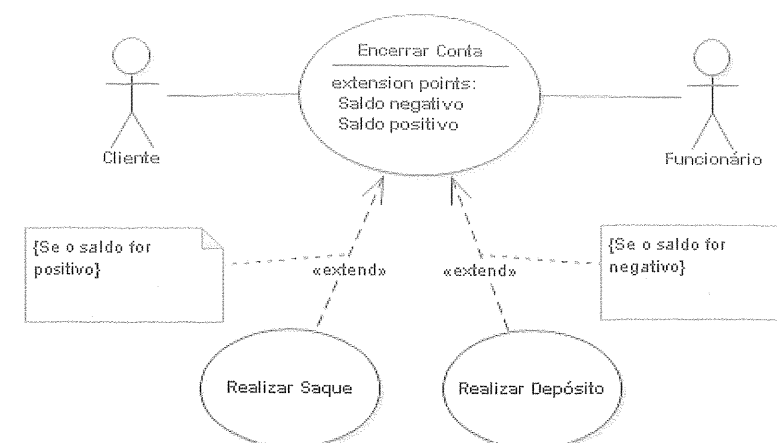


Figura 3.14 – Exemplo de Ponto de Extensão.

Tabela 3.2 – Documentação do Caso de Uso Encerrar Conta

Nome do Caso de Uso	Encerrar Conta
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	Funcionário
Resumo	Este caso de uso descreve as etapas necessárias para que um cliente encerre uma conta
Pré-Condições	É necessário existir uma conta ativa
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Solicitar encerramento de conta fornecendo o número da conta em questão	
	2. Executar caso de uso Emitir Saldo
	3. Encerrar a conta
Restrições/Validações	1. A conta só pode ser encerrada pelo seu titular 2. A conta só pode ser encerrada se o seu saldo estiver zerado
Fluxo Alternativo I – Saldo Positivo	
Ações do Ator	Ações do Sistema
	1. Executar Caso de Uso Realizar Saque
Fluxo Alternativo II – Saldo Negativo	
Ações do Ator	Ações do Sistema
1. Fornecer valor para depósito	
	2. Executar Caso de Uso Realizar Depósito
Fluxo Alternativo III – Manutenção do Cadastro do Cliente	
Ações do Ator	Ações do Sistema
	1. Se for a única conta do cliente, atualizar seu cadastro, tornando-o inativo – Executar Caso de Uso Manter Cliente.

3.10 Multiplicidade no Diagrama de Casos de Uso

A multiplicidade em uma associação entre um ator e um caso de uso basicamente especifica o número de vezes que um ator pode utilizar um determinado caso de uso. A figura 3.15 apresenta um exemplo de multiplicidade em associações entre atores e casos de uso.



Figura 3.15 – Exemplos de Multiplicidade em Associações entre Atores e Casos de Uso.

Ao examinarmos esse exemplo, percebemos que um ator **Sócio** utiliza o caso de uso **Cadastrar Sócio** somente uma vez, enquanto o ator **Funcionário** pode utilizá-lo muitas vezes. Da mesma forma, percebemos que esse caso de uso só pode ser utilizado por um sócio e um funcionário por vez.

3.11 Estereótipos

Os estereótipos possibilitam certo grau de extensibilidade aos componentes ou associações da UML, além de permitir a identificação de componentes ou associações que, embora semelhantes aos outros, tenham alguma característica que os diferenciem, dando-lhes mais destaque no diagrama. Estereótipos podem atribuir funções extras ou diferentes a um componente, permitindo que este possa ser utilizado para modelar situações diferentes das quais foi originalmente projetado. Existe uma grande quantidade de estereótipos que podem ser aplicados aos mais diversos componentes da UML e o engenheiro de software pode criar seus próprios estereótipos se assim desejar, como será explicado ao final deste livro.

Como exemplo, poderíamos querer deixar bem claro que o caso de uso **Abrir Conta** refere-se a um processo. Assim, poderíamos atribuir-lhe o estereótipo `<<process>>`, cujo objetivo é deixar explícito que o caso de uso em questão refere-se a um processo. Esse tipo de estereótipo é apresentado na parte superior do componente, acima da descrição de seu nome. Existem alguns estereótipos que simplesmente apresentam um texto entre sinais de maior e menor sobre o componente, chamados estereótipos de texto e outros que modificam o desenho-padrão do componente, chamados de estereótipos gráficos. A figura 3.16 ilustra um exemplo de estereótipo de texto. Exemplos de estereótipos gráficos serão apresentados no capítulo sobre o diagrama de classes.



Figura 3.16 – Estereótipo de texto.

3.12 Fronteira de Sistema

Uma fronteira de sistema identifica um classificador que contém um conjunto de casos de uso. Ela permite identificar um subsistema ou mesmo um sistema completo, além de destacar o que está contido no sistema e o que não está. Atores são externos ao sistema enquanto casos de uso são internos. Uma fronteira de sistema é representada por um retângulo envolvendo os casos de uso por ela contidos, além de um título que a descreve. Ao longo deste capítulo serão apresentados diversos exemplos contendo fronteiras de sistema.

3.13 Exemplo de Diagrama de Casos de Uso – Sistema de Controle Bancário

A figura 3.17 apresenta o diagrama de casos de uso referente a um sistema de controle bancário. Esse sistema permite que seus clientes abram e encerrem contas, bem como depositem ou saquem valores e emitam saldos ou extratos. Essas últimas quatro funcionalidades o cliente pode utilizar diretamente por meio de um caixa eletrônico, porém, para abrir ou encerrar uma conta ele necessitará interagir com um funcionário do banco, que poderá ainda realizar alguma manutenção em seu cadastro, ou seja, registrá-lo ou alterar seus dados.

Pode-se perceber que um retângulo intitulado **Sistema de Controle Bancário** envolve todos os casos de uso do diagrama. Esse retângulo representa a fronteira do sistema, cujo objetivo, como já foi dito, é separar os atores externos das funcionalidades do software. Os atores, externos ao sistema, representam os clientes e funcionários do banco.

No sistema representado pelo diagrama, primeiramente um cliente, representado aqui por um ator, solicita a abertura de uma conta, a qual pode ser uma conta comum, que não permite a retirada de mais dinheiro do que está depositado, uma conta especial, que permite o saque extra até um determinado limite, ou uma conta-poupança, que rende juros enquanto o dinheiro depositado permanecer sem ser movimentado.

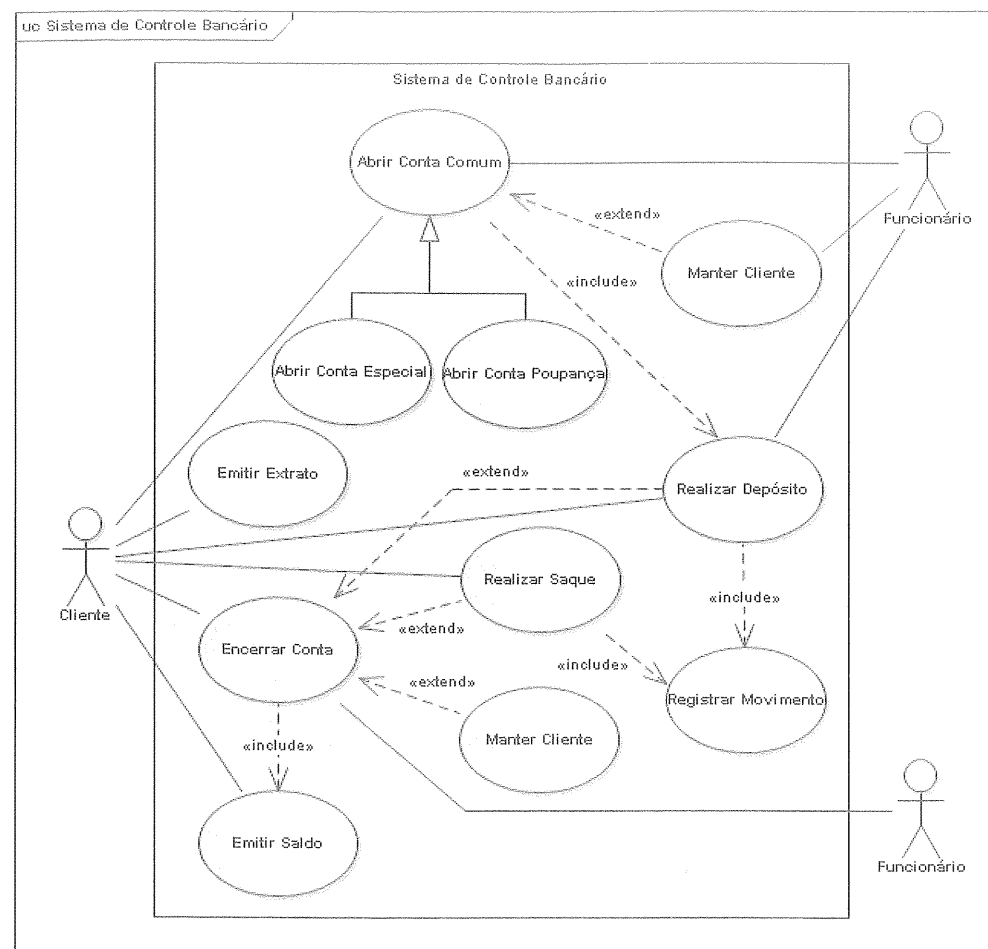


Figura 3.17 – Diagrama de Casos de Uso – Sistema de Controle Bancário.

Como os processos de abertura para cada tipo de conta têm características muito semelhantes entre si, com poucas distinções, resolveu-se colocar o caso de uso **Abrir Conta Comum** como uma generalização (embora ele seja efetivamente utilizado) e os outros dois tipos de abertura como especializações do primeiro, detalhando-se em sua documentação as características particulares que elas têm em relação ao caso de uso geral **Abrir Conta Comum**. Uma alternativa seria definir um caso de uso geral, denominado simplesmente **Abrir Conta**, que não seria utilizado diretamente, e derivar a partir daí os três tipos de abertura de conta possíveis.

Podemos perceber que existe uma associação do tipo extensão entre o caso de uso **Abrir Conta Comum** (herdada também por suas especializações) e o **Manter Clientes**. Embora o caso de uso **Manter Clientes** possa ser utilizado independentemente pelos funcionários do banco, a criação de uma conta bancária

normalmente implica o registro do novo cliente ou, se este já estiver cadastrado, uma possível atualização. No entanto, como nem sempre é necessário registrar ou atualizar o cliente, esse caso de uso não constitui uma inclusão, pois só será utilizado se o cliente que deseja abrir a conta não estiver registrado ou precisar atualizar seus dados.

Há também uma associação do tipo inclusão entre o caso de uso **Abrir Conta Comum** (igualmente herdado por suas especializações) e o **Realizar Depósito**, porque é obrigatório depositar algum valor no momento em que o processo de abertura da conta for concluído, o que exige uma associação do tipo inclusão com o caso de uso **Realizar Depósito**. Assim, sempre que uma conta for aberta, as instruções contidas no caso de uso **Realizar Depósito** serão igualmente executadas.

Um cliente pode eventualmente querer encerrar uma conta, porém, antes de efetuar o encerramento, algumas operações devem ser levadas a efeito. Em primeiro lugar, é preciso verificar o saldo da conta para determinar se o banco precisa devolver algum dinheiro ao cliente ou, caso a conta seja especial e estiver negativa, se o cliente precisa depositar algum dinheiro para encerrar a conta. A verificação do saldo é obrigatória, por isso, utiliza uma associação do tipo inclusão entre o caso de uso **Encerrar Conta** e o caso de uso **Emitir Saldo**. Se o saldo estiver positivo, deve-se realizar um saque por meio do caso de uso **Realizar Saque**. Se o saldo estiver negativo, deve-se realizar um depósito por intermédio do caso de uso **Realizar Depósito**. Como não é possível saber se vai ser necessário ou não sacar ou depositar algum valor, as associações entre os casos de uso **Encerrar Conta**, **Realizar Saque** e **Realizar Depósito** são representadas por extensões.

Esse diagrama representa ainda uma última associação de extensão entre o caso de uso **Encerrar Conta** e o **Manter Cliente**. Essa associação é necessária devido à possibilidade de a conta a ser encerrada constituir-se na única conta mantida pelo cliente, o que determina a manutenção do registro do mesmo, identificando-o como inativo, pois, uma vez cliente, seu registro não pode ser excluído do sistema, o mesmo ocorrendo com quaisquer contas que um dia ele possa ter tido na instituição bancária. Esta pode ser definida como encerrada, mas jamais excluída.

Existem ainda os casos de uso **Emitir Saldo** e **Emitir Extrato**, que são serviços que podem ser utilizados diretamente pelo cliente por meio de um caixa eletrônico, sem a intermediação de um funcionário. Como são serviços simples, não necessitam de interações com outros casos de uso do diagrama, embora o caso de uso **Emitir Saldo** seja utilizado pelo caso de uso **Encerrar Conta**, conforme já descrito.

Finalmente, temos os casos de uso **Realizar Saque** e **Realizar Depósito**, que já foram citados, por poderem ser também requisitados pelo caso de uso **Encerrar Conta**. Normalmente, no entanto, tais serviços, da mesma maneira que os casos de uso **Saldo** e **Extrato**, serão utilizados diretamente pelo cliente. Contudo, esses serviços necessitam registrar o movimento realizado, razão pela qual ambos obrigatoriamente utilizam o caso de uso **Registrar Movimento**, que, para fins de histórico bancário, registra qualquer saque ou depósito porventura realizado em uma conta. Por ser obrigatória sua utilização, a associação entre os casos de uso **Realizar Saque** e **Realizar Depósito** com o caso de uso **Registrar Movimento** precisa ser uma associação de inclusão.

O exemplo do sistema de controle bancário é relativamente complexo, utilizando diversos tipos de associações. No entanto, deve-se ter em mente que o diagrama de casos de uso é muito geral e tem por objetivo principal oferecer uma visão externa do sistema ao usuário, caracterizando-se muitas vezes por ser bastante simples. Assim, deve-se evitar desenvolver diagramas de casos de uso muito complexos sempre que possível. Ao longo deste capítulo e nas soluções dos exercícios propostos apresentaremos novos exemplos, algumas vezes mais simples que este.

3.14 Documentação do Diagrama de Casos de Uso do Sistema de Controle Bancário

Nesta seção daremos uma sugestão de como documentar um diagrama de casos de uso. Os atores e casos de uso aqui documentados referem-se ao diagrama de casos de uso do sistema de controle bancário apresentado na figura 3.17, descrita na seção anterior. Os casos de uso **Abrir Conta** e **Encerrar Conta** já foram apresentados anteriormente nas seções anteriores.

3.14.1 Atores que Interagem com o Sistema

- **Cliente** – Este ator representa as pessoas físicas ou jurídicas que mantêm ou mantiveram contas na instituição bancária, com direito a utilizar serviços como saque, depósito, saldo ou extrato enquanto mantinham contas ativas.
- **Funcionário** – Este ator representa os funcionários contratados para atender presencialmente os clientes da instituição. São basicamente os caixas do banco.

3.14.2 Documentação do Caso de Uso Abrir Conta Especial

Tabela 3.3 – Documentação do Caso de Uso Abrir Conta Especial

Nome do Caso de Uso	Abrir Conta Especial
Caso de Uso Geral	Abrir Conta Comum
Ator Principal	Cliente
Atores Secundários	Funcionário
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para abrir uma conta corrente especial
Pré-Condições	O pedido de abertura precisa ser aprovado
Pós-Condições	É necessário realizar um depósito inicial
Fluxo Principal	
Ações do Ator	Ações do Sistema
Idênticas às do caso de uso Abrir Conta Comum, exceto por fornecer comprovante de estar empregado e de que seu salário é superior a R\$ 500,00	Idênticas às do caso de uso Abrir Conta Comum, exceto por definir o limite do cheque especial após a aprovação do pedido de abertura
Restrições/Validações	1. Para abrir uma conta corrente é preciso ser maior de idade
	2. É necessário comprovar estar empregado e o salário tem que ser superior a R\$ 500,00
	3. O valor mínimo de depósito inicial é R\$ 25,00

3.14.3 Documentação do Caso de Uso Abrir Conta Poupança

Tabela 3.4 – Documentação do Caso de Uso Abrir Conta Poupança

Nome do Caso de Uso	Abrir Conta Poupança
Caso de Uso Geral	Abrir Conta Comum
Ator Principal	Cliente
Atores Secundários	Funcionário
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para abrir uma conta poupança
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
Idênticas as do caso de uso Abrir Conta Comum	Idênticas as do caso de uso Abrir Conta Comum, exceto por definir a data de aniversário da conta, que não necessariamente será a data de abertura da conta, uma vez que a abertura de uma conta poupança não obriga um depósito
Restrições/Validações	

3.14.4 Documentação do Caso de Uso Manter Clientes

Tabela 3.5 – Documentação do Caso de Uso Manter Clientes

Nome do Caso de Uso	Manter Cliente
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	
Resumo	Este caso de uso descreve as possíveis atividades de manutenção do cadastro de clientes, ou seja, permite incluir, alterar ou consultar clientes. Um cliente não pode ser excluído apenas tornado inativo
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar o CPF ou CNPJ do cliente	
	2. Consultar o cliente por seu CPF ou CNPJ
	3. Se já houver um cliente cadastrado com o CPF ou CNPJ informados, apresentar seus dados
4. Se necessário, alterar ou inserir os dados do cliente.	
	5. Se necessário, gravar as atualizações
Restrições/Validações	1. O CPF ou CNPJ precisam ser validados 2. Os campos nome, endereço e data de nascimento são obrigatórios

3.14.5 Documentação do Caso de Uso Emitir Saldo

Tabela 3.6 – Documentação do Caso de Uso Emitir Saldo

Nome do Caso de Uso	Emitir Saldo
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Descreve os passos necessários para um cliente obter o saldo referente a uma determinada conta
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar o número da conta	
	2. Verificar a existência da conta
	3. Solicitar a senha da conta
4. Informar a senha	
	5. Verificar se a senha está correta

	6. Emitir o saldo
Restrições/Validações	1. A conta precisa existir e estar ativa 2. A senha precisa estar correta
Fluxo de Exceção I – Conta não encontrada	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que o número da conta informada não foi encontrado
Fluxo de Exceção II – Senha inválida	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que a senha fornecida não combina com a senha da conta

3.14.6 Documentação do Caso de Uso Emitir Extrato

Tabela 3.7 – Documentação do Caso de Uso Emitir Extrato

Nome do Caso de Uso	Emitir Extrato
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Descreve os passos necessários para um cliente obter o extrato referente a uma determinada conta em um determinado período
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar o número da conta	
	2. Verificar se a conta existe e está ativa
	3. Solicitar a senha
4. Informar a senha	
	5. Verificar se a senha está correta
	6. Solicitar períodos
7. Informar o período inicial e final do extrato	
	8. Listar os movimentos da conta dentro do período informado
Restrições/Validações	1. A conta precisa existir e estar ativa 2. A senha precisa estar correta 3. Os períodos precisam estar corretos e o período inicial não pode ser maior que o final
Fluxo de Exceção – Períodos inválidos	
Ações do Ator	Ações do Sistema
	1. Informar ao cliente que as datas fornecidas são inválidas

3.14.7 Documentação do Caso de Uso Realizar Depósito

Tabela 3.8 – Documentação do Caso de Uso Realizar Depósito

Nome do Caso de Uso	Realizar Depósito
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	Funcionário
Resumo	Descreve os passos necessários para um cliente depositar algum valor em uma determinada conta
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar o número da conta	
	2. Verificar se a conta existe
3. Fornecer o valor a ser depositado	
	4. Executar caso de uso Registrar Movimento
Restrições/Validações	1. A conta precisa existir e estar ativa

3.14.8 Documentação do Caso de Uso Realizar Saque

Tabela 3.9 – Documentação do Caso de Uso Realizar Saque

Nome do Caso de Uso	Realizar Saque
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Descreve os passos necessários para um cliente sacar algum valor de uma determinada conta
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar o número da conta	
	2. Verificar se a conta existe
	3. Solicitar a senha
4. Informar a senha	
	5. Verificar se a senha está correta
6. Informar o valor a ser retirado	
	7. Se o valor solicitado for válido, entregar a importância ao cliente
	8. Executar caso de uso Registrar Movimento

Restrições/Validações	1. A conta precisa existir e estar ativa 2. A senha precisa estar correta
Fluxo Alternativo I – Conta Comum/Poupança	
Ações do Ator	Ações do Sistema
	1. Se o valor solicitado for igual ou menor que o saldo da conta, sacar valor
Restrições/Validações	1. O valor a ser retirado precisa ser igual ou menor que o saldo da conta
Fluxo Alternativo II – Conta Especial	
Ações do Ator	Ações do Sistema
	1. Se o valor solicitado for igual ou menor que o saldo da conta somado ao limite, sacar valor
Restrições/Validações	1. O valor a ser retirado precisa ser igual ou menor que o saldo somado ao limite da conta
Fluxo de Exceção – Saldo insuficiente	
Ações do Ator	Ações do Sistema
	1. Se o valor solicitado for superior ao que o cliente pode sacar, emitir uma mensagem informando que o saldo é insuficiente e recusar o pedido

3.14.9 Documentação do Caso de Uso Registrar Movimento

Tabela 3.10 – Documentação do Caso de Uso Registrar Movimento

Nome do Caso de Uso	Registrar Movimento
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Descreve os passos necessários para registrar um movimento referente a um saque ou a um depósito
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
	1. Receber o número da conta movimentada, o tipo do movimento (se é depósito ou saque), a data e o valor movimentado
	2. Registrar o movimento
Restrições/Validações	

3.15 Exemplo de Diagrama de Casos de Uso – Sistema de Videolocadora

Nesta seção apresentamos o diagrama de casos de uso para um sistema de videolocadora, considerando que a locadora tem cópias de muitos filmes dos mais variados gêneros, sendo necessário, portanto, existir um módulo para manutenção do cadastro de filmes. Uma vez que pode ser interessante poder pesquisar filmes por gênero ou por determinado ator, torna-se também necessária a existência de um módulo para manutenção de gêneros e outro para manutenção do cadastro de atores. Da mesma forma, só é permitido locar cópias para sócios registrados, sendo, portanto, necessária a existência de um módulo para a manutenção do cadastro de sócios da videolocadora.

O processo principal desse sistema é a locação de cópias propriamente dita, na qual o sócio se identifica e informa as cópias que deseja locar. Se o registro do sócio existir e este não possuir cópias em atraso, a locação será autorizada. Finalmente, existe ainda um processo importante para a empresa: a emissão dos filmes mais locados, para se ter uma ideia da preferência dos clientes. A figura 3.18 apresenta o diagrama de casos de uso para esse sistema.

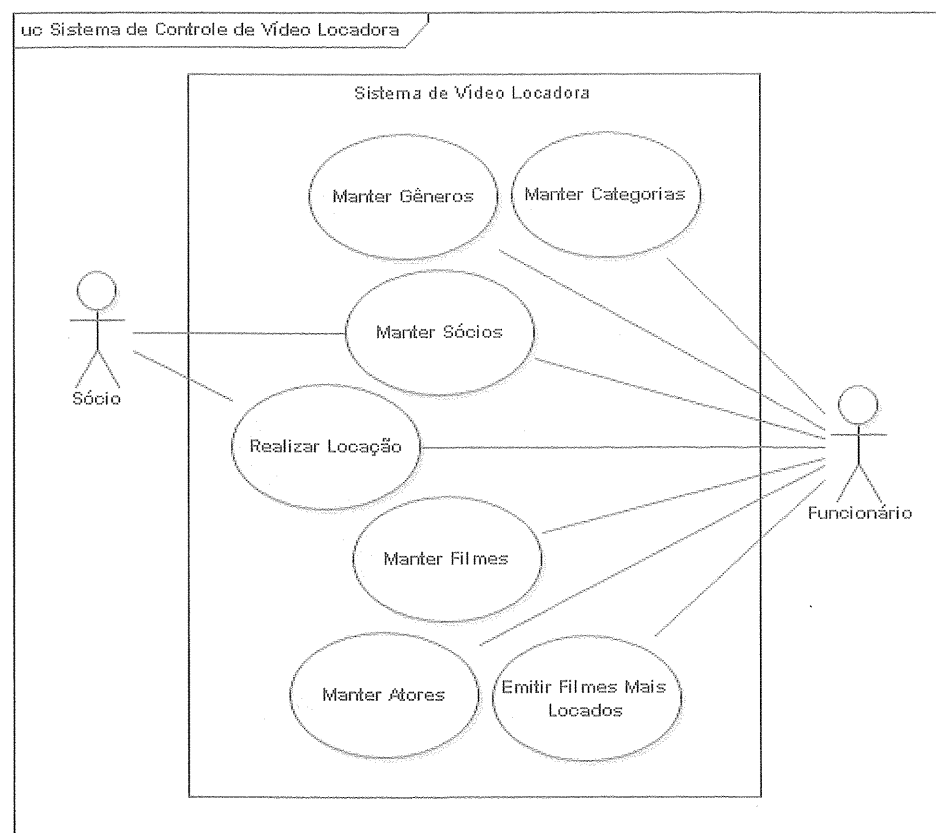


Figura 3.18 – Diagrama de Casos de Uso – Sistema de Videolocadora.

Esse diagrama representa cada processo de manutenção de cadastro como um caso de uso e o identifica como **Manter Sócios** ou **Manter Filmes**. Na verdade, estes são casos de uso secundários, que servem como apoio aos casos de uso primários representados aqui basicamente pelo caso de uso **Realizar Locação**, que representa o processo de locação de cópias de filmes por um sócio. O caso de uso **Emitir Filmes Mais Locados** também é um caso de uso secundário, uma vez que consiste basicamente em um relatório que apresenta os filmes locados pela ordem de maior preferência. Em sistemas mais complexos onde haja muitos módulos de manutenção de pequena importância, não é muito recomendado inserir um caso de uso para cada um desses módulos, uma vez que o diagrama se tornará muito poluído, contendo muitas informações de pouca importância.

Observe que o sócio somente interage com os casos de uso **Manter Sócios**, já que ele fornece as informações ao funcionário para a manutenção de seu cadastro, bem como com o caso de uso **Realizar Locação**, uma vez que ele é o ator mais interessado no processo, identificando-se e informando quais cópias de filmes deseja locar. Os outros casos de uso são acessados somente pelo funcionário.

O leitor talvez venha a se perguntar onde é feita a verificação para determinar se o sócio já se encontra cadastrado ou se esse sócio possui locações pendentes. Todas as etapas estão contidas no próprio caso de uso **Realizar Locação**, já que não há necessidade de criar um caso de uso exclusivo para qualquer uma dessas validações, sendo perfeitamente possível defini-las na documentação do caso de uso.

3.16 Exemplo de Diagrama de Casos de Uso – Sistema de Telefone Celular

Nesta seção apresentaremos o diagrama de casos de uso para um sistema de telefone celular levando em consideração os seguintes requisitos:

- O celular oferece o serviço de realizar chamadas, no qual o usuário deve informar um telefone para que o celular ligue. O celular deve registrar as últimas chamadas.
- Semelhante ao serviço de chamadas, o telefone oferece o serviço de mensagens, onde o usuário deve informar o número de telefone para o qual deseja enviar a mensagem. O celular deve igualmente registrar as últimas mensagens.
- O aparelho oferece o serviço de agenda, a partir do qual é possível cadastrar os diversos contatos do usuário. Cada contato armazena o nome do contato e seu telefone. Caso o usuário consulte um telefone já existente, ele poderá ligar para esse contato ou enviar uma mensagem. O sistema deve guardar as últimas ligações feitas, bem como as últimas mensagens enviadas.

- O celular oferece também o serviço de recebimento de chamadas. O sistema deve avisar o recebimento de uma chamada por meio do toque de uma música, e o usuário pode aceitar a chamada ou não. As últimas ligações também devem ser gravadas.
- Da mesma forma, o sistema deve oferecer o serviço de recebimento de mensagens, devendo também registrar as últimas mensagens recebidas.
- O celular oferece ainda o serviço de despertador, no qual o usuário pode cadastrar e/ou ativar um ou mais horários para despertar.
- Finalmente, o sistema oferece o serviço de tons, no qual o usuário pode selecionar, entre muitas músicas possíveis, a que mais lhe agrada para avisar-lhe do recebimento de uma chamada ou mensagem, ou para despertá-lo.

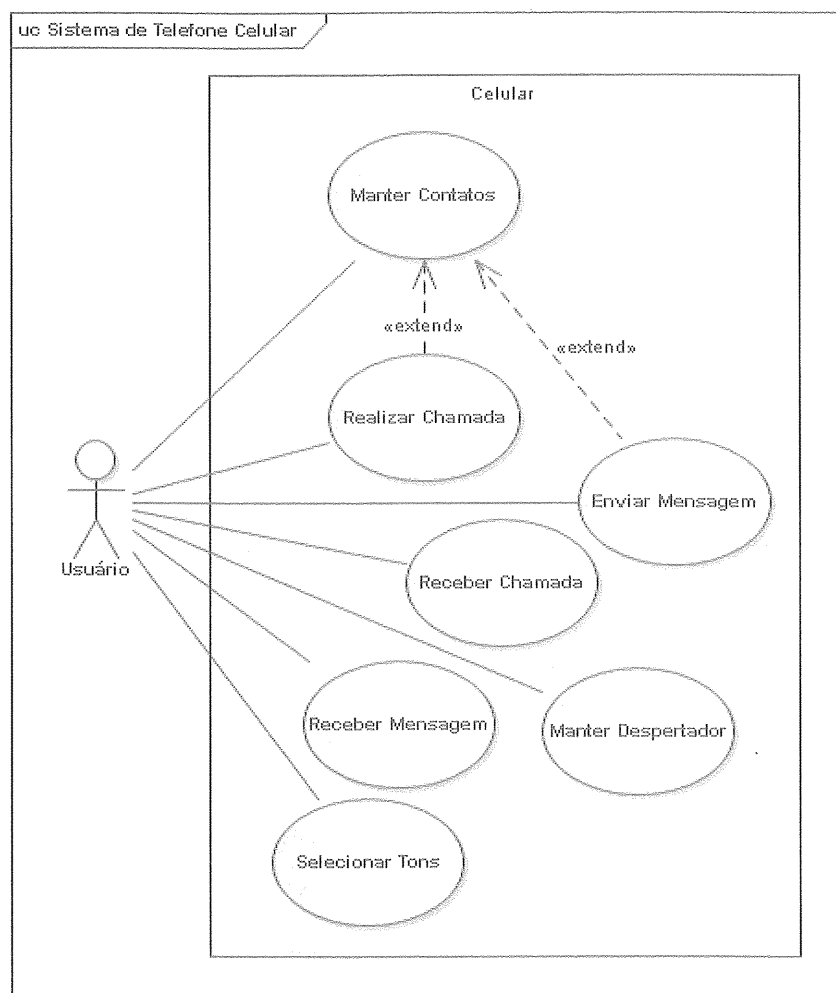


Figura 3.19 – Diagrama de Casos de Uso – Sistema de Telefone Celular.

Nesta solução, o único ator representado é o usuário, uma vez que somente ele interage com o aparelho, não havendo interação com nenhum outro ator. A seguir detalharemos os casos de uso identificados:

- **Realizar Chamada** – este caso de uso representa o processo, bem simples, aliás, pelo qual o usuário faz uma chamada para um número de telefone, bastando para isso informar o número do telefone e pressionar o botão para chamá-lo, sendo que o sistema deverá registrar cada chamada efetuada, registrando o número e se ela foi atendida ou não. Se o limite máximo de chamadas registradas for atingido, o sistema deverá excluir a mais antiga.
- **Enviar Mensagem** – este caso de uso representa os passos necessários para que o usuário envie uma mensagem para outro telefone celular. Esse processo é quase tão simples quanto o anterior, no qual o usuário deve informar o número para o qual deseja enviar a mensagem, digitá-la e pressionar o botão para enviá-la. O celular deverá também registrar essa mensagem e, da mesma forma que no processo anterior, se o limite máximo de mensagens registradas for atingido, o sistema deverá excluir a mais antiga.
- **Receber Chamada** – este caso de uso representa o processo pelo qual o usuário recebe uma chamada. Quando esse evento ocorre, o aparelho deverá avisar o usuário por meio de uma música, por exemplo. Cabe ao usuário optar por atendê-la ou não. De qualquer forma, o celular deverá registrá-la, identificando se foi atendida ou recusada.
- **Receber Mensagem** – este caso de uso representa o processo pelo qual o usuário recebe uma mensagem, cujo evento deverá ser também notificado pelo aparelho e registrado por ele, excluindo a mensagem mais antiga se o número máximo de mensagens recebidas já tiver sido atingido.
- **Manter Contatos** – este caso de uso representa os passos que o usuário deve percorrer para inserir um novo elemento na agenda ou consultar um elemento já existente. Se quiser inserir um novo número, deverá informar o nome da pessoa/empresa que ele deseja registrar, bem como o número a ela pertencente. Caso tenha consultado um número da agenda, poderá alterá-lo ou realizar uma chamada ou enviar uma mensagem a partir do número consultado. Por esse motivo, existem os relacionamentos de extensão entre esse caso de uso e os casos de uso **Realizar Chamada** e **Enviar Mensagem**.
- **Manter Despertador** – este caso de uso representa os passos necessários para que o usuário insira, altere ou exclua horários em que o celular deverá despertá-lo.

- **Selecionar Tons** – Finalmente, este caso de uso representa o processo por meio do qual o usuário seleciona os tons de música que mais lhe agradam ouvir quando receber uma chamada, uma mensagem ou o despertador tocar. O usuário deverá selecionar qual dessas opções deseja alterar o tom de música e escolher em uma lista a música que considerar mais adequada.

3.17 Exemplo de Diagrama de Casos de Uso – Sistema de Clínica Veterinária

Neste exemplo, o sistema de clínica veterinária a ser modelado se comporta da seguinte maneira:

- Os clientes primeiramente marcam consultas com a secretária, fornecendo suas informações pessoais e as dos animais que desejam tratar. Se o cliente ou o animal ainda não estiver cadastrado no sistema ou existir algum dado que precise ser atualizado, a secretária deverá atualizar o cadastro.
- Em cada sessão de tratamento (uma sessão equivale a uma consulta), o cliente deve informar os sintomas aparentes do animal, os quais devem ser registrados. Um tratamento pode ser encerrado em apenas uma consulta, quando se tratar de algo simples, ou pode arrastar-se por muitas sessões, dependendo do diagnóstico do médico-veterinário.
- Durante uma consulta, o veterinário pode marcar exames para o animal, a serem trazidos na sessão seguinte. O pedido dos exames e seus resultados devem ser registrados no histórico de tratamentos do animal. Após cada sessão, o histórico da consulta deve ser atualizado.
- É responsabilidade da secretária manter atualizados os cadastros de clientes, animais, médicos e espécies. A figura 3.20 apresenta a solução para esse sistema na forma de um diagrama de casos de uso.

Os atores que compõem esse diagrama são descritos a seguir:

- **Cliente** – este Ator representa uma pessoa física que possua um ou mais animais que alguma vez foram tratados pela clínica.
- **Secretária** – a descrição deste ator é autoexplicativa. Ele representa os funcionários da clínica responsáveis por marcar consultas e manter a maioria dos cadastros da empresa.
- **Veterinário** – este ator também é autoexplicativo, representando os médicos-veterinários da clínica que atendem os animais.

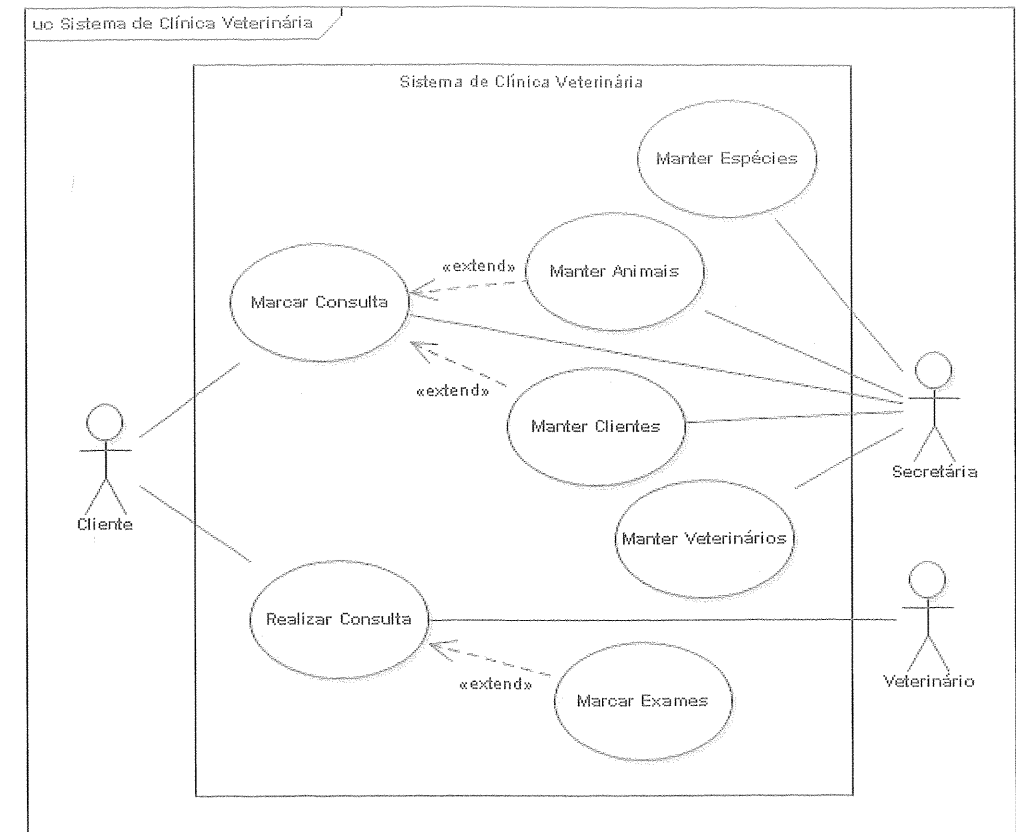


Figura 3.20 – Diagrama de Casos de Uso – Sistema de Clínica Veterinária.

Os casos de uso que compõem o sistema são os seguintes:

- **Marcar Consulta** – este caso de uso representa as etapas necessárias para que um cliente possa agendar uma consulta para um determinado animal. Nesse caso de uso interagem os atores **Cliente** e **Secretária**. Observe que, a partir desse caso de uso, podem ser executados os casos de uso **Manter Clientes** e **Manter Animais**, devido à possibilidade de que seja necessário registrar um novo cliente ou animal ou de que seus dados precisem ser atualizados. Observe que o ator **Secretária** pode utilizar esses dois últimos casos de uso, independentemente do caso de uso **Marcar Consulta**, como é possível verificar por meio da associação direta entre o ator **Secretária** e esses casos de uso.
- **Manter Veterinários** e **Manter Espécies** – estes dois casos de uso são bastante simples, representando os módulos de cadastro dos veterinários que trabalham na clínica, bem como as espécies de animais que são tratados na veterinária.
- **Realizar Consulta** – este caso de uso representa a documentação da consulta propriamente dita, iniciando a ser preenchida logo no início da consulta e

sendo finalizada ao término da mesma pelo médico-veterinário responsável. Observe que inserimos um relacionamento de extensão com o caso de uso **Marcar Exames**, já que eventualmente o veterinário pode pedir ao cliente para realizar exames no animal em questão. Na verdade, a documentação desse último caso de uso poderia estar no próprio caso de uso **Realizar Consulta**. Preferimos, no entanto, representá-lo de forma separada para tornar mais clara a compreensão do diagrama. Obviamente, os resultados dos exames seriam verificados e registrados na consulta seguinte.

3.18 Exemplo de Diagrama de Casos de Uso – Sistema de Controle de Advocacia

A seguir apresentaremos o diagrama de casos de uso para um sistema de controle de processos jurídicos, de acordo com as seguintes afirmações:

- Um cliente, pessoa física ou jurídica, que paga um advogado para defendê-lo ou para processar outra pessoa, procura este profissional. Se o cliente ainda não estiver cadastrado, o advogado deverá registrar seus dados pessoais.
- Em seguida, o cliente deve fornecer informações a respeito do processo que deseja que o advogado mova contra alguém ou que o defenda de outra pessoa. Obviamente, o processo precisa ser registrado e receberá diversas adições enquanto estiver em andamento. O cliente deve fornecer também informações sobre a parte contrária (pessoa física ou jurídica que está processando ou sendo processada pelo cliente), que deverá também ser registrada, caso ainda não esteja. Observe que uma mesma pessoa física ou jurídica pode ser tanto um cliente como uma parte contrária em períodos diferentes, obviamente.
- Um processo deve tramitar em um determinado tribunal e em uma determinada vara. No entanto um tribunal pode julgar muitos processos, e uma vara pode ter diversos processos tramitando nela. Um tribunal pode ter inúmeras varas, porém, um processo julgado por um determinado tribunal só pode tramitar em uma das varas pertencentes ao mesmo. O advogado pode achar necessário emitir relatórios de todos os processos em andamento em um determinado tribunal e tramitando em uma determinada vara.
- Cada processo tem no mínimo uma audiência, e cada audiência relativa a um determinado processo deve conter sua data e a recomendação do tribunal. Para fins de histórico do processo, cada audiência deve ser registrada.
- Um processo pode gerar custas (despesas com fotocópias, viagens etc.). Cada custo deve ser armazenada de forma a ser cobrada da parte contrária, caso o processo seja ganho.

- Esse sistema deve estar integrado a um sistema de contas a pagar e receber; cada custo gera uma conta a pagar. Caso o processo seja ganho, ele irá gerar uma ou mais contas a receber, dependendo da negociação com a parte contrária.

A figura 3.21 apresenta uma solução para o problema enunciado por meio de um diagrama de casos de uso. Esse diagrama é manipulado pelos seguintes atores:

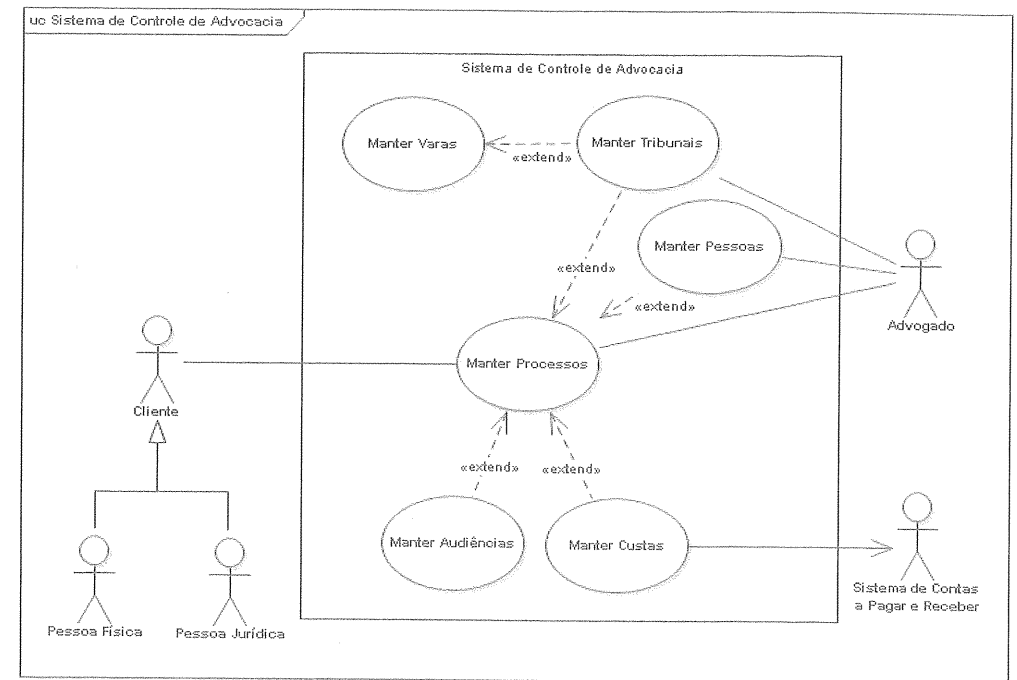


Figura 3.21 – Diagrama de Casos de Uso – Sistema de Controle de Advocacia.

- **Cliente** – ator que representa as pessoas físicas ou jurídicas que solicitam que o advogado as defenda ou processe outra pessoa. Observe que existem duas especializações a partir do ator **Cliente**, chamadas **Pessoa Física** e **Pessoa Jurídica**, indicando que o cliente pode tanto constituir-se de uma pessoa física quanto de uma pessoa jurídica.
- **Advogado** – este ator representa os advogados do escritório de advocacia responsáveis pela interação com o sistema.
- **Sistema de Contas a Pagar e Receber** – este ator representa o sistema integrado ao de controle de processos para onde são enviadas as custas geradas por um determinado processo.

A seguir descreveremos os casos de uso que compõem esse diagrama:

- **Manter Processos** – este é o principal caso de uso desse sistema, no qual são manipulados os processos em andamento do escritório. Note que, além do ator **Advogado**, o ator **Cliente** também interage com este caso de uso, não diretamente, é claro, mas ele precisa fornecer as informações sobre o processo ao advogado. A esse caso de uso estão ligados praticamente todos os demais casos de uso do diagrama.
- **Manter Pessoas** – este caso de uso representa o módulo de manutenção de pessoas que fazem ou já fizeram parte de algum processo, seja como clientes, seja como partes contrárias. O **Advogado** pode manter qualquer pessoa a qualquer momento. No entanto, esse caso de uso está ligado ao caso de uso **Manter Processo**, pois pode ser chamado a partir dele sempre que um cliente ou parte contrária ainda não estiver registrado ou seus dados necessitarem de atualização. Como a chamada a esse caso de uso só ocorre nas situações citadas, a associação entre esses casos de uso é classificada como uma extensão.
- **Manter Tribunais e Manter Varas** – estes casos de uso representam os módulos de manutenção de tribunais e varas. O caso de uso **Manter Varas** só pode ser chamado a partir do caso de uso **Manter Tribunais**. Na verdade, o caso de uso **Manter Varas** poderia estar contido no caso de uso **Manter Tribunais**, porém resolvemos apresentá-lo separadamente para facilitar a compreensão do diagrama. Observe que o caso de uso **Manter Tribunais** pode tanto ser utilizado de forma independente quanto ser chamado a partir do caso de uso **Manter Processos**, na eventual possibilidade de ser necessária alguma alteração do tribunal ou vara onde o processo for tramitar.
- **Manter Audiências** – conforme já foi dito anteriormente, um processo tem no mínimo uma audiência, podendo transcorrer ao longo de diversas delas. Essas audiências precisam ser registradas. Assim, criamos um caso de uso responsável pela manutenção das audiências de cada processo. A associação de extensão entre os casos de uso **Manter Audiências** e **Manter Processos** indica que o caso de uso apenas será executado eventualmente a partir do processo ao qual ele pertence, ou seja, sempre que uma nova audiência for marcada, esta deve ser registrada, e sempre que uma audiência tenha transcorrido, a mesma deve ser atualizada com a recomendação do tribunal.
- **Manter Custas** – um processo normalmente envolve despesas que são conhecidas como custas. Cada possível custo de um determinado processo precisa ser registrada. Observe que da mesma forma que o caso de uso **Manter Audiências**, o caso de uso **Manter Custas** só poderá ser chamado a partir do caso de uso **Manter Processos**, e mesmo assim somente quando houver necessidade. Por isso, novamente utilizamos um relacionamento

de extensão entre os dois casos de uso. Note que outro ator interage com o caso de uso em questão, o ator denominado **Sistema de Contas a Pagar e Receber**, que representa um sistema integrado que registra cada custo de um processo, como uma conta a receber. É importante notar a seta da associação em questão, que demonstra que o ator não envia informações, somente recebe.

3.19 Exercícios Propostos

3.19.1 Sistema de Controle de Cinema

Desenvolva o diagrama de casos de uso para um sistema de controle de cinema, sabendo que:

- Um cinema pode ter muitas salas, sendo necessário, portanto, registrar informações a respeito de cada uma, como sua capacidade, ou seja, o número de assentos disponíveis.
- O cinema apresenta muitos filmes. Um filme tem informações como título e duração. Assim, sempre que um filme for ser apresentado, deve-se registrá-lo também.
- Um mesmo filme pode ser apresentado em diferentes salas e em horários diferentes. Cada apresentação em uma determinada sala e horário é chamada Sessão. Um filme sendo apresentado em uma sessão tem um conjunto máximo de ingressos, determinado pela capacidade da sala.
- Os clientes do cinema podem comprar ou não ingressos para assistir a uma sessão. O funcionário deve intermediar a compra do ingresso. Um ingresso deve conter informações como o tipo de ingresso (meio ingresso ou ingresso inteiro). Além disso, um cliente só pode comprar ingressos para sessões ainda não encerradas.

3.19.2 Sistema de Controle de Clube Social

Desenvolva um diagrama de casos de uso para um sistema de controle de clube social de acordo com os seguintes requisitos:

- Para ingressar em um clube é necessário apresentar uma solicitação, a ser avaliada por uma comissão nomeada pelo clube.
- Em caso de aprovação, o candidato pode associar-se no clube. Opcionalmente, caso possua dependentes, poderá associá-los também, o que obviamente aumentará o valor da mensalidade a ser paga.

- Uma vez sendo sócio do clube, deverá pagar uma mensalidade para poder frequentá-lo.
- As mensalidades são geradas pelo clube levando em consideração a categoria do sócio e o número de seus dependentes.

3.19.3 Sistema de Locação de Veículos

Desenvolva o diagrama de casos de uso para um sistema de controle de aluguel de veículos, levando em consideração os seguintes requisitos:

- A empresa tem uma grande frota de carros de passeio, sendo que esses carros apresentam diferentes marcas e modelos. Eventualmente um carro pode ser retirado da frota devido a algum acidente grave ou simplesmente por ter sido considerado velho demais para o padrão da empresa e tenha sido vendido. Da mesma forma, a empresa eventualmente renova a frota, sendo necessário, portanto, estar sempre mantendo o cadastro de veículos da empresa.
- Os clientes dirigem-se à empresa e solicitam o aluguel de carros. No entanto, primeiramente é necessário cadastrá-los, caso ainda não possuam cadastro ou seus dados tenham sido alterados.
- Depois de ter se identificado/cadastrado, o cliente escolherá o carro que deseja alugar (o valor da locação varia de acordo com o ano, marca e modelo do automóvel). Durante o processo de locação, o cliente deve informar por quanto tempo utilizará o carro, para qual finalidade e por onde desejará trafegar, já que essas informações também influenciam o preço da locação. Antes de liberar o veículo, a empresa exige que o cliente forneça um valor superior ao estabelecido na análise da locação, a título de caução. Caso o cliente não utilize todo o valor da caução até o momento da devolução do veículo, o valor restante lhe será devolvido.
- Quando o cliente devolve o carro deve-se definir o automóvel como devolvido, registrar a data e hora da devolução e a quilometragem em que se encontra, bem como verificar se o automóvel se encontra nas mesmas condições em que foi alugado. Caso o cliente tenha ocupado o carro por mais tempo que o combinado, deve pagar o aluguel referente ao tempo extra em que permaneceu com veículo. Da mesma maneira, o cliente deverá pagar por qualquer dano sofrido pelo veículo quando este encontrava-se locado. Por outro lado, o cliente pode ser ressarcido de parte do valor que pagou caso o custo do tempo em que esteve de posse do veículo seja inferior ao valor previamente fornecido.

3.19.4 Sistema para Controle de Leilão Via Internet

Desenvolva o diagrama de casos de uso para um sistema de leilão via internet, de acordo com os seguintes requisitos:

- Existem diversos participantes em cada leilão, interessados em adquirir os itens ofertados. Os participantes devem se registrar via internet, antes de o leilão iniciar.
- Durante o leilão são ofertados cada um dos itens que estão arrolados.
- Um participante pode realizar quantos lances quiser durante a realização do leilão, mas não é obrigado a realizar lance nenhum. Antes de poder fazer quaisquer ofertas, ele precisa se logar no sistema.
- Sempre que um lance suplantar o lance anterior, o sistema deve anunciá-lo, declarando qual o vencedor quando os lances se encerrarem.

3.19.5 Sistema de Controle de Hotelaria

Desenvolva o diagrama de casos de uso para um sistema de controle de hotelaria sabendo que:

- Os quartos podem ser alugados no momento em que o hóspede chega ao hotel (desde que existam vagas) ou serem reservados via internet.
- Caso seja a primeira vez que aluga quartos, ou seus dados tenham mudado, o hóspede deve ser cadastrado antes de finalizar o aluguel do quarto.
- Além do aluguel do quarto, o hotel oferece diversos serviços, como restaurante, lavar e/ou passar roupas etc. Obviamente, qualquer desses serviços, se solicitado, será cobrado na fatura final.
- O hóspede pode também consumir os produtos contidos no frigobar, que também são cobrados pelo hotel.
- As diárias vencem ao meio-dia. A política do hotel exige que as diárias sejam quitadas semanalmente. Quando o cliente for quitar a fatura, quitará não somente as diárias do(s) quarto(s) que alugou, mas também qualquer serviço que tenha solicitado e os itens consumidos no frigobar.
- O hóspede, depois de quitar a fatura, pode permanecer no hotel ou encerrar sua estadia.
- Quando for encerrar sua estadia, o hóspede deverá pagar quaisquer serviços e/ou diárias ainda não pagas.

3.20 Resolução dos Exercícios

3.20.1 Sistema de Controle de Cinema

A figura 3.22 apresenta a solução desse exercício. Os atores identificados foram:

- **Funcionário** – a função desse ator é óbvia. Ele representa os funcionários que trabalham no cinema e são responsáveis por vender os ingressos e realizar a manutenção nos cadastros.
- **Cliente** – já este ator representa as pessoas que assistirão às sessões de filmes oferecidas pelo cinema.

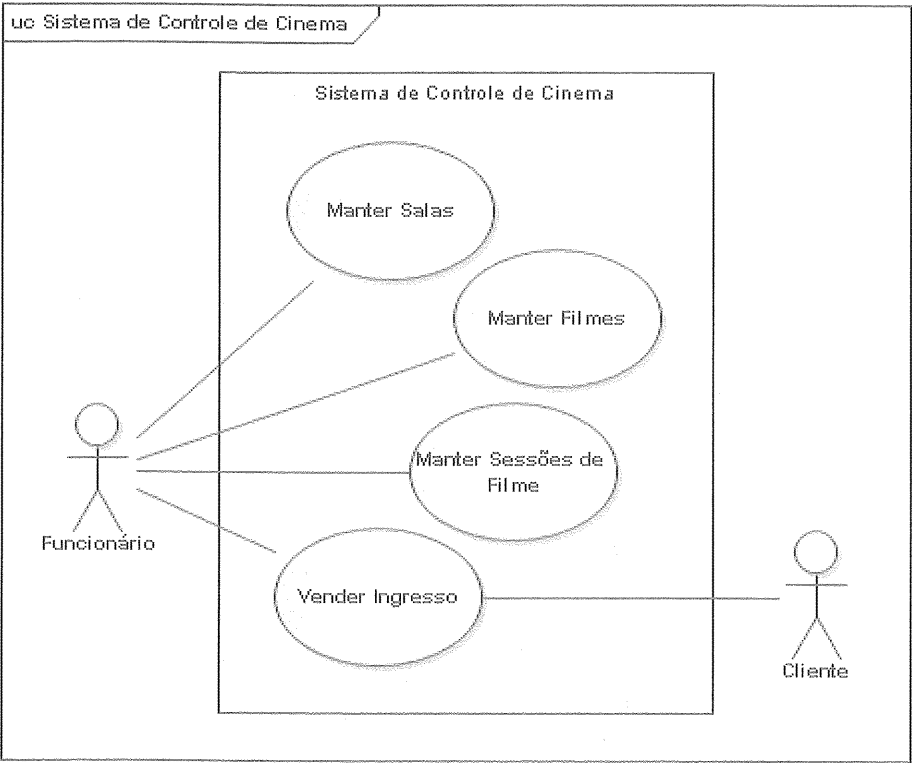


Figura 3.22 – Diagrama de Casos de Uso – Sistema de Controle de Cinema.

- A seguir descreveremos os casos de uso que compõem esse diagrama:
- **Manter Salas** – este é um caso de uso secundário que se refere ao processo de manutenção das salas cadastradas no sistema.
 - **Manter Filmes** – este caso de uso, também secundário, refere-se ao processo de manutenção do cadastro do acervo de filmes adquiridos pelo cinema.

- **Manter Sessões de Filmes** – este caso de uso, igualmente secundário, representa a manutenção do cadastro de sessões, onde são definidos que filmes serão apresentados em quais salas e em que datas e horários.
- **Vender Ingresso** – este é o principal caso de uso desse sistema, sendo o único primário, e apresenta os passos necessários para que o funcionário venda um ingresso para uma sessão a um cliente.

Como forma de ilustrar o processo de venda de ingressos, apresentamos a seguir, na tabela 3.11, a documentação do caso de uso **Vender Ingresso**.

Tabela 3.11 – Documentação do Caso de Uso Vender Ingresso

Nome do Caso de Uso	Vender Ingresso
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	Cliente
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário para emitir um ingresso para uma sessão de cinema
Pré-Condições	Devem haver assentos disponíveis
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Venda de Ingresso	
	2. Apresentar sessões disponíveis
3. Informar sessão desejada e se deseja um ingresso inteiro ou meio ingresso	
	4. Emitir ingresso
Restrições/Validações	

3.20.2 Sistema de Controle de Clube Social

A figura 3.23 apresenta a solução do exercício de clube social. Os atores identificados foram:

- **Candidato** – este ator representa uma pessoa interessada em associar-se ao clube, mas cujo pedido precisa ser aprovado.
- **Sócio** – este ator, derivado a partir do ator **Candidato**, representa o sócio propriamente dito, depois de seu cadastro ter sido aprovado.
- **Clube** – este ator representa os funcionários que interagem com os sócios do clube e manipulam funções do sistema.

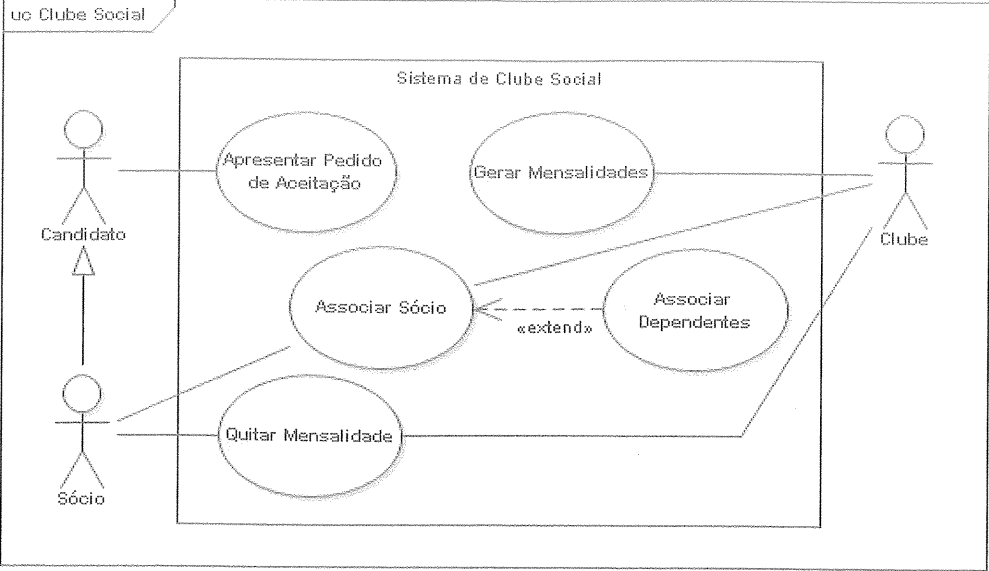


Figura 3.23 – Diagrama de Casos de Uso – Sistema de Clube Social.

A seguir descreveremos os casos de uso que compõem esse diagrama:

- **Apresentar Pedido de Aceitação** – este caso de uso refere-se ao processo por meio do qual um candidato a sócio apresenta seu pedido para ser aceito como sócio do clube.
- **Associar Sôcio** – uma vez tendo seu pedido aprovado, este caso detalha os passos necessários para que um funcionário do clube associe um novo sócio. Observe que há uma interação entre o ator Sôcio com esse caso de uso, uma vez que ele fornece os dados necessários para que a associação seja realizada. Observe ainda que existe uma associação de extensão entre esse caso de uso e o caso de uso **Associar Dependentes**, onde são detalhados as etapas pelas quais o sócio poderá associar seus possíveis dependentes. Levando-se em consideração que esse caso de uso é opcional, já que o sócio não necessariamente terá dependentes, essa associação caracteriza-se por ser uma extensão.
- **Gerar Mensalidades** – este caso de uso representa o processo para gerar as mensalidades dos sócios do clube e é manipulado exclusivamente por um funcionário.
- **Quitar Mensalidade** – este é o processo pelo qual o sócio interage com um funcionário para quitar uma ou mais mensalidades.

A tabela 3.12 apresenta a documentação do processo de Quitar Mensalidade.

Tabela 3.12 – Documentação do Caso de Uso Quitar Mensalidade

Nome do Caso de Uso	Quitar Mensalidade
Caso de Uso Geral	
Ator Principal	Sócio
Atores Secundários	Funcionário
Resumo	Este caso de uso descreve as etapas percorridas por um sócio para quitar uma ou mais mensalidades
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar ao funcionário que deseja quitar mensalidades	
2. O funcionário seleciona a opção quitar mensalidades	
	3. Solicitar número do cartão do sócio
4. Informar cartão	
	5. Consultar sócio
	6. Consultar mensalidades
	7. Calcular juros das mensalidades (se necessário)
	8. Apresentar mensalidades
9. O sócio seleciona as mensalidades a pagar e realizar o pagamento	
10. O funcionário solicita a quitação das mensalidades selecionadas	
	11. Quitar mensalidades
	12. Emitir recibo
Fluxo de Exceção	
Ações do Ator	Ações do Sistema
	1. Caso o cartão não seja válido emitir mensagem de erro
Restrições/Validações	

3.20.3 Sistema de Locação de Veículos

A figura 3.24 apresenta a solução do exercício de sistema de locação de veículos. Os atores identificados foram:

- **Cliente** – este ator representa os clientes que desejam locar veículos na locadora. Esse ator interage com todos os casos de uso, uma vez que o funcionário precisa de informações do cliente para utilizá-los, sendo a

- única exceção o caso de uso Manter Veículos, manipulado exclusivamente pelo funcionário.
- **Funcionário** – este ator representa os funcionários que atendem os clientes da empresa.

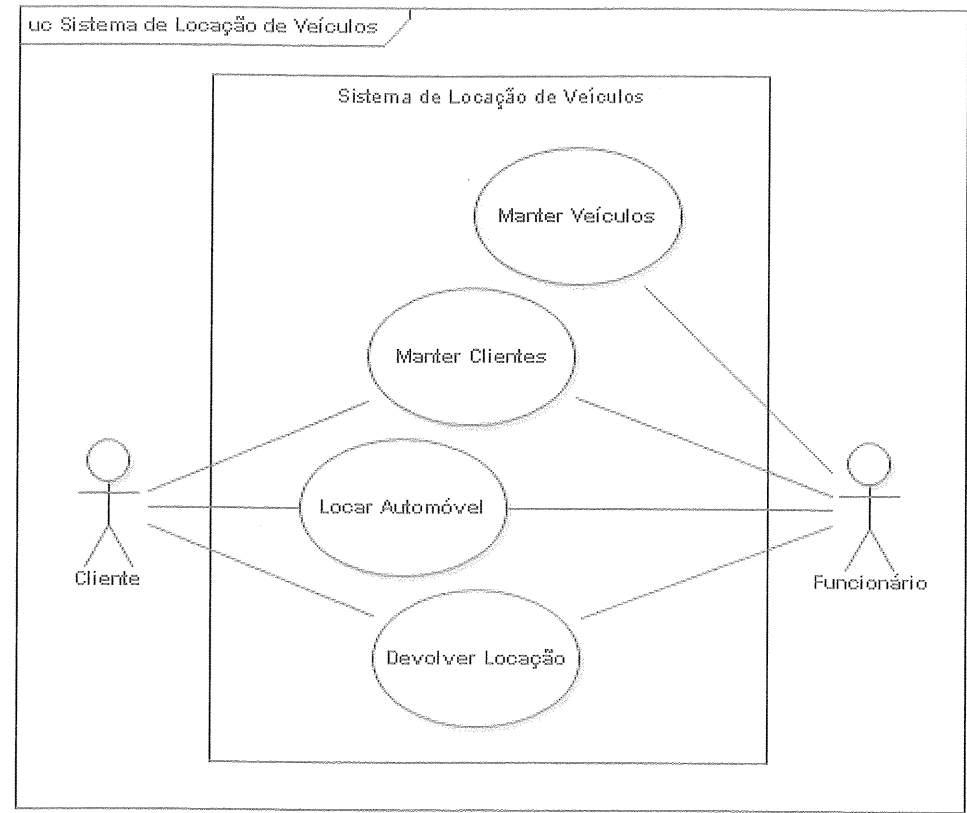


Figura 3.24 – Diagrama de Casos de Uso – Sistema de Controle de Aluguel de Carros.

A seguir descreveremos os casos de uso que compõem esse diagrama:

- **Manter Veículos** – este é um caso de uso secundário que representa o processo de manutenção do cadastro de veículos da empresa.
- **Manter Clientes** – este é também um processo secundário e representa a manutenção do cadastro de clientes. Sempre que um novo cliente solicitar a locação de um veículo, se este ainda não estiver registrado ou seus dados tiverem sido alterados, o funcionário deverá executar esse caso de uso.
- **Locar Automóvel** – este caso de uso identifica as etapas necessárias para que um cliente consiga locar um automóvel. É necessário que ele selecione o veículo que deseja locar e informar por quanto tempo deseja locá-lo, bem como para qual finalidade e por onde deseja trafegar, além de fornecer um valor de caução para poder alugar o automóvel.

- **Devolver Locação** – este caso de uso identifica os passos que serão executados quando o usuário devolver o veículo, onde será registrada a data e a hora da devolução do automóvel, sua quilometragem e se este encontra-se nas mesmas condições de quando foi alugado. Nesse processo o cliente pode ter que pagar o aluguel referente ao período extra que ocupou o veículo ou qualquer dano ou multa sofrida enquanto utilizava o mesmo. Por outro lado, ele pode vir a ser ressarcido de parte do valor que pagou se tiver ocupado o automóvel por menos tempo.

A tabela 3.13 apresenta a documentação do processo de Locar Automóvel.

Tabela 3.13 – Documentação do Caso de Uso Locar Automóvel

Nome do Caso de Uso	Locar Automóvel
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	Cliente
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário para realizar a locação de um automóvel
Pré-Condições	O cliente precisa estar cadastrado
Pós-Condições	Receber valor de caução
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção de locação de veículos	
	2. Carregar clientes
	3. Carregar veículos disponíveis
4. Selecionar cliente	
5. Selecionar veículo	
	6. Apresentar detalhes do automóvel
7. Informar dados locação	
	8. Calcular valor da locação e valor da caução
9. Fornecer valor da caução	
	10. Registrar locação
Restrições/Validações	

3.20.4 Sistema para Controle de Leilão Via Internet

A figura 3.25 apresenta a solução do exercício referente ao sistema de leilão via internet. Os atores identificados foram:

- **Participante** – este ator representa as pessoas interessadas em participar do leilão, registrando-se previamente antes de este iniciar e que, se assim desejarem, podem fazer lances e eventualmente arrematar algum item ofertado.
- **Leiloeiro** – este ator representa o funcionário responsável por manipular o cadastro de leilões, bem como registrar os itens a serem leiloados, determinando seus lances mínimos. Além disso, esse ator representa o funcionário que coordenará o leilão, dando-lhe início, oferecendo os itens, controlando os lances e os participantes que os fizerem, bem como anunciando os itens que forem arrematados e por quem. É também responsabilidade desse ator encerrar o leilão.

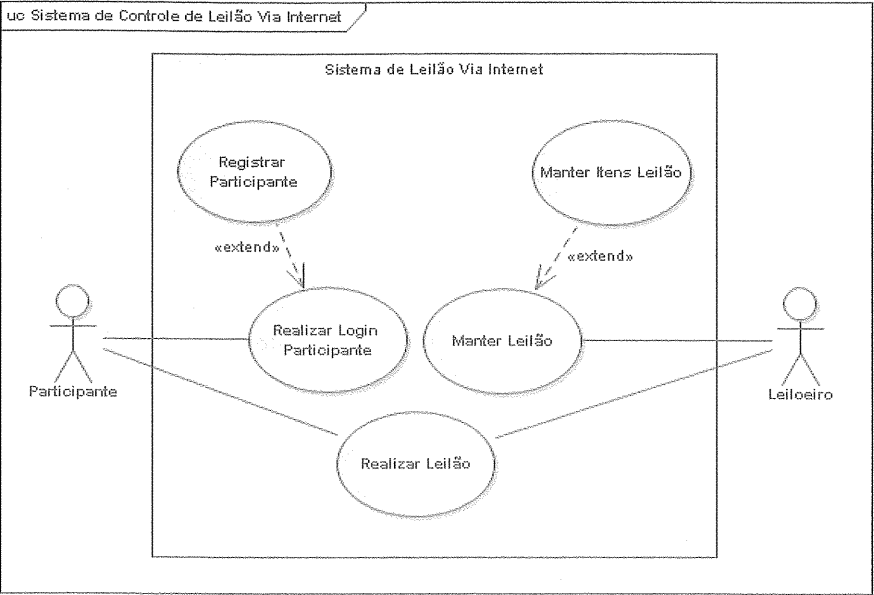


Figura 3.25 – Diagrama de Casos de Uso – Sistema de Controle de Leilão Via Internet.

A seguir descreveremos os casos de uso que compõem esse diagrama:

- **Manter Leilão** – este é um caso de uso secundário que representa o processo para a manutenção dos leilões agendados, podendo-se incluir um novo leilão ou modificar a data de início de um leilão já registrado, por exemplo. Observe que há um relacionamento de extensão entre esse caso de uso e o caso de uso **Manter Itens Leilão**, uma vez que é necessário primeiro consultar um leilão para depois dar manutenção nos itens a serem leiloados neste.
- **Realizar Login Participante** – este é o caso de uso que estabelece as etapas para que um participante logue-se no sistema e possa participar do leilão.
- **Registrar Participante** – este é um caso de uso secundário que define os passos percorridos para que o cliente se registre, caso não possua um nome-login e uma senha para poder participar de um leilão.

- **Realizar Leilão** – Este é o caso de uso principal do sistema, que representa as etapas percorridas pelo leiloeiro para abrir o leilão, ofertar seus itens, receber os lances dos participantes, anunciar os vencedores e encerrar o leilão. O ator participante também interage com esse caso de uso, uma vez que é ele quem poderá fazer algum lance em relação aos itens ofertados. A tabela 3.14 apresenta a documentação do processo de Realizar Leilão.

Tabela 3.14 – Documentação do Caso de Uso Realizar Leilão

Nome do Caso de Uso	Realizar Leilão
Caso de Uso Geral	
Ator Principal	Leiloeiro
Atores Secundários	Participante
Resumo	Este caso de uso descreve as etapas percorridas por um leiloeiro para realizar um leilão
Pré-Condições	O participante precisa estar logado
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. O leiloeiro seleciona a opção realizar leilão	
	2. Apresentar leilões em aberto
3. O Leiloeiro seleciona e inicia um leilão	
	4. Apresentar itens do leilão
5. [Enquanto houver itens] o Leiloeiro seleciona e anuncia o item a leiloar	
6. Se assim o desejar, o Participante oferece um lance	
	7. Registrar lance
8. Caso tenha havido ao menos um lance, o Leiloeiro anuncia o participante vencedor	
	9. Arrematar item
10. [Quando não houver mais itens a leiloar] o Leiloeiro finaliza o leilão	
	11. Finalizar leilão
Fluxo Alternativo	
Ações do Ator	Ações do Sistema
	1. Caso um item não receba nenhum lance, ele não será arrematado
Restrições/Validações	Os participantes não são obrigados a realizar lances

3.20.5 Sistema de Controle de Hotelaria

A figura 3.26 apresenta a solução do exercício referente ao sistema de controle de hotelaria. Os atores identificados foram:

- **Hóspede** – este ator representa, como o próprio nome diz, os hóspedes que se hospedam no hotel.
- **Funcionário** – este ator representa os funcionários responsáveis por reservar e alugar quartos, bem como manter o cadastro de hóspedes, realizar serviços e quitar diárias.

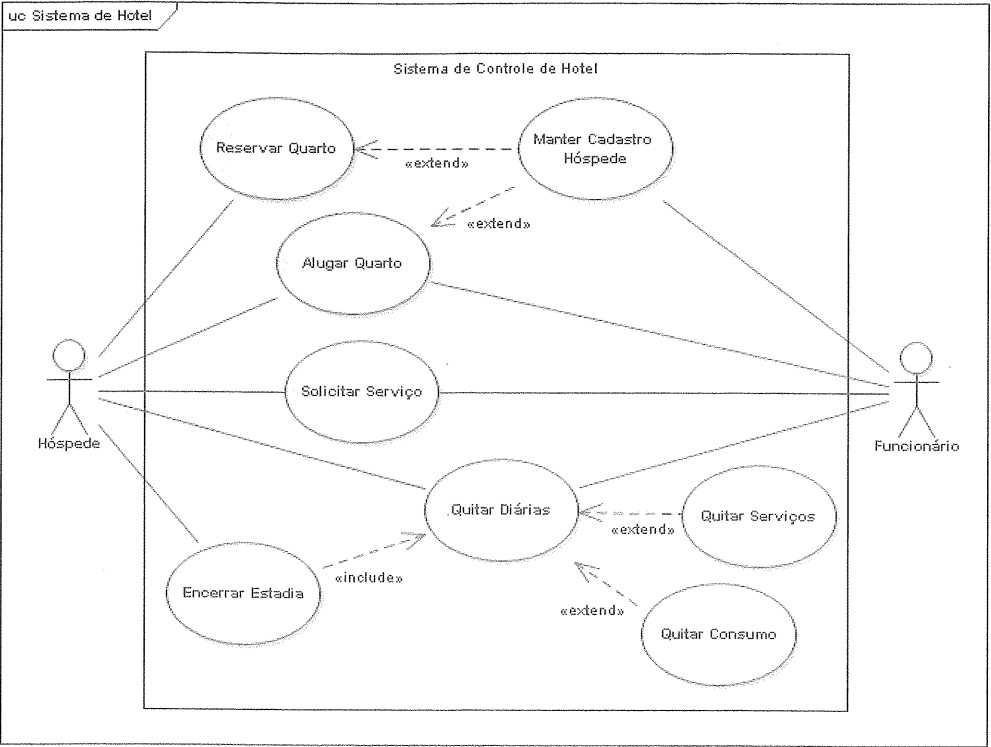


Figura 3.26 – Diagrama de Casos de Uso – Sistema de Controle de Hotelaria.

A seguir descreveremos os casos de uso que compõem este diagrama:

- **Reservar Quarto** – este caso de uso representa o processo pelo qual um hóspede reserva um ou mais quartos no hotel.
- **Alugar Quarto** – este caso de uso representa o processo pelo qual um hóspede aluga um ou mais quartos no hotel.

- **Manter Cadastro Hóspede** – este caso de uso representa as etapas necessárias para que um funcionário efetue a manutenção do cadastro de hóspedes. Observe que existe uma associação de extensão entre esse caso de uso e os casos de uso **Reservar Quarto** e **Alugar Quarto**, uma vez que, caso o hóspede não possua cadastro no hotel ou seus dados tenham sofrido alguma alteração desde a última reserva ou aluguel, é necessário registrar ou alterar seu cadastro.
- **Solicitar Serviço** – este caso de uso apresenta as etapas necessárias para que o hóspede solicite algum dos serviços oferecidos pelo hotel.
- **Quitar Diárias** – as regras do hotel exigem que as diárias sejam pagas semanalmente. Este caso de uso apresenta os passos percorridos por um hóspede para quitar suas diárias. A quitação das diárias não necessariamente significa o encerramento da estadia do hóspede. Na quitação das diárias é necessário quitar também quaisquer serviços solicitados ou consumo do frigobar. Por esse motivo existe a associação de extensão entre esse caso de uso e os casos de uso **Quitar Serviços** e **Quitar Consumo**, onde estão detalhados os passos para que o hóspede possa saldar essas dívidas.
- **Encerrar Estadia** – este caso de uso representa o processo pelo qual um hóspede encerra sua estadia no estabelecimento. Observe que há um relacionamento de inclusão com o caso de uso **Quitar Diárias**, uma vez que ao encerrar sua estadia o hóspede deverá quitar as diárias ainda não pagas.

A tabela 3.15 apresenta a documentação do processo de Quitar Diárias.

Tabela 3.15 – Documentação do Caso de Uso Quitar Diárias

Nome do Caso de Uso	Quitar Diárias
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	Hóspede
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário durante o processo de quitação de diárias por um hóspede
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar quarto e solicitar quitação de diárias	
	2. Selecionar e apresentar diárias
3. Selecionar quitar diárias	
	4. Quitar diárias

Fluxo Alternativo I	
Ações do Ator	Ações do Sistema
	1. Caso tenham sido solicitados serviços executar o caso de uso Quitar Serviços
Fluxo Alternativo II	
Ações do Ator	Ações do Sistema
	1. Caso tenha havido consumo de itens do frigobar executar o caso de uso Quitar Consumo
Restrições/Validações	



CAPÍTULO 4

Diagrama de Classes

O diagrama de classes é um dos mais importantes e mais utilizados da UML. Seu principal enfoque está em permitir a visualização das classes que comporão o sistema com seus respectivos atributos e métodos, bem como em demonstrar como as classes do diagrama se relacionam, complementam e transmitem informações entre si. Esse diagrama apresenta uma visão estática de como as classes estão organizadas, preocupando-se em como definir a estrutura lógica das mesmas. O diagrama de classes serve ainda como base para a construção da maioria dos outros diagramas da linguagem UML.

Basicamente, o diagrama de classes é composto por suas classes e pelas associações existentes entre elas, ou seja, os relacionamentos entre as classes. Alguns métodos de desenvolvimento de software, como o Processo Unificado, recomendam que se utilize o diagrama de classes ainda durante a fase de análise, produzindo-se um modelo conceitual a respeito das informações necessárias ao software. No modelo conceitual, o engenheiro preocupa-se apenas em representar as informações que o software necessitará, em termos de classes e seus atributos, bem como as associações entre as classes, não modelando características como os métodos que as classes poderão conter nessa etapa (os métodos já fazem parte do “como” o software será desenvolvido). Somente na fase de projeto toma-se o modelo conceitual do diagrama de classes e produz-se o modelo de domínio, que já enfoca a solução do problema. Os métodos necessários às classes são descobertos a partir da modelagem de diagramas de interação, como o diagrama de sequência, que será estudado nos próximos capítulos.

4.1 Atributos e Métodos

Classes costumam ter atributos, que, como já explicado no capítulo 2, armazenam os dados dos objetos da classe, além de métodos, também chamados operações, que são as funções que uma instância da classe pode executar. Os valores dos atributos podem variar de uma instância para outra. Graças a essa

característica, aliás, é possível identificar cada objeto individualmente, ao passo que os métodos são idênticos para todas as instâncias de uma classe específica.

Embora os métodos sejam declarados no diagrama de classes, identificando os possíveis parâmetros que são por eles recebidos e os possíveis valores por eles retornados, o diagrama de classes não se preocupa em definir as etapas que tais métodos deverão percorrer quando forem chamados, sendo esta uma função atribuída a outros diagramas, como o diagrama de atividade, que será analisado nos capítulos seguintes. A figura 4.1 apresenta um exemplo de classe contendo atributos e métodos.

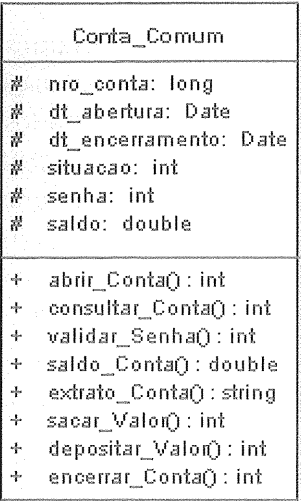


Figura 4.1 – Classe.

Como já foi explicado no capítulo 2, uma classe, na linguagem UML, é representada como um retângulo com até três divisões, descritas a seguir:

- A primeira contém a descrição ou nome da classe, que nesse exemplo é `Conta_Comum`.
- A segunda armazena os atributos e seus tipos de dados (o formato que os dados devem ter para serem armazenados em um atributo). No exemplo da figura 4.1 a classe `Conta_Comum` contém os atributos `nro_conta`, do tipo `long`; `dt_abertura` e `dt_encerramento`, do tipo `Date` (este não é um tipo primitivo e se refere a uma classe); `situacao` e `senha`, do tipo `int`; e `saldo`, do tipo `double`.
- Finalmente, a terceira divisão lista os métodos da classe. Nesse exemplo a classe `Conta_Comum` contém os métodos `abrir_Conta`, `consultar_Conta`, `validar_Senha`, `saldo_Conta`, `extrato_Conta`, `sacar_Valor`, `depositar_Valor` e `encerrar_Conta`.

Os símbolos de sustenido (#) e mais (+) na frente dos atributos e métodos representam a visibilidade dos mesmos, o que determina quais objetos de quais classes podem utilizar o atributo ou o método em questão. Os tipos de visibilidade já foram explicados no capítulo 2.

Não é realmente obrigatório que uma classe apresente as três divisões, pois pode haver classes que não tenham atributos ou que não contenham métodos, ou pode acontecer ainda que seus atributos e métodos não precisem ser apresentados no diagrama, já que é recomendado apresentar apenas atributos relevantes ao diagrama para evitar, por exemplo, tornar o diagrama muito poluído. Assim, é possível encontrar classes com somente duas divisões ou mesmo com apenas uma, no caso, aquela que contém a descrição da classe, porque esta é obrigatória.

Métodos podem receber valores como parâmetros e retornar valores (da mesma forma que as funções implementadas na linguagem de programação C), que podem tanto ser o resultado produzido pela execução do método quanto simplesmente um valor representado se o método foi realizado com sucesso ou não. Nessa classe, por exemplo, podemos perceber que o retorno do método `abrir_Conta` é um `long`, que conterà o número da nova conta gerada. Por sua vez, o retorno dos métodos `consultar_Conta`, `validar_Senha`, `sacar_Valor`, `depositar_Valor` e `encerrar_Conta` é um inteiro (`int`), e esse valor é utilizado para determinar se o método foi concluído com sucesso ou não. Por padrão o retorno 1 significa verdadeiro (ou sucesso) e 0 falso (ou fracasso). Já os métodos `saldo_Conta` e `extrato_Conta` retornam um `double` e uma `String`, que contêm o resultado da execução desses métodos. Para o método `abrir_Conta`, se o valor retornado for igual a 1 significará que o método foi concluído com sucesso e que uma nova conta foi aberta. Já se o retorno for igual a zero, saberemos que o método não foi concluído da forma esperada e que algum problema ocorreu.

Na figura 4.1 os métodos foram apresentados sem o detalhamento de quais argumentos (parâmetros) eles deveriam receber (a lista de argumentos de um método, junto com seu valor de retorno, é chamada de assinatura da operação). Esse detalhamento é opcional, e isso foi feito propositadamente para explicar os diagramas que serão apresentados no decorrer do capítulo. Alguns métodos podem ter muitos parâmetros, e o detalhamento destes em um diagrama que contenha muitas classes tornará esse diagrama muito extenso e será difícil visualizá-lo claramente como um todo, porque as classes ficarão largas. Assim, é melhor, quando se trata de apresentar um diagrama de classes composto por muitas classes, apresentar somente o nome dos métodos da classe, sem especificar os argumentos que ele receberá. O detalhamento dos métodos de cada classe poderá ser feito individualmente em um diagrama separado, conforme apresentado na figura 4.2.

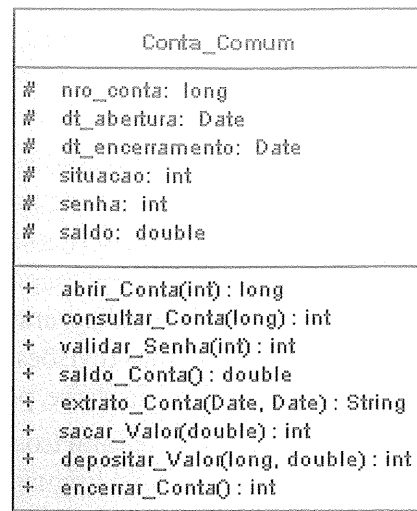


Figura 4.2 – Detalhamento das Assinaturas das Operações.

Ao examinarmos a figura 4.2 podemos perceber que os métodos `abrir_Conta` e `validar_Senha` recebem um inteiro como parâmetro; o método `consultar_Conta`, um `long`; `sacar_Valor`, um `double`; e `depositar_Valor`, um `long` e um `double`. Já os outros métodos não recebem argumento algum. A partir do modelo da classe `conta_Comum` apresentada na figura 4.2 é possível gerar o código correspondente a ele. A seguir, será apresentado o código correspondente a essa classe implementado em Java.

```

public class Conta_Comum {
    protected long nro_conta;
    protected Date dt_abertura;
    protected Date dt_encerramento;
    protected int situacao;
    protected int senha;
    protected double saldo;

    public Conta_Comum() {
    }
    public void finalize() throws Throwable {
    }
    public long abrir_Conta(int senha) {
        return 0;
    }
    public int consultar_Conta(long nro_conta) {
        return 0;
    }
}
  
```

```

    public int validar_Senha(int senha) {
        return 0;
    }
    public double saldo_Conta() {
        return 0;
    }
    public String extrato_Conta() {
        return "";
    }
    public int sacar_Valor(double valor) {
        return 0;
    }
    public int depositar_Valor(long nro_conta, double valor) {
        return 0;
    }
    public int encerrar_Conta() {
        return 0;
    }
}
  
```

Obviamente esse código é apenas um esqueleto da classe. Como pode-se perceber, não há nenhum detalhamento de como os métodos irão desempenhar sua função. Além disso, será preciso criar uma classe chamada `Date` para que o compilador reconheça os atributos desse tipo. Observe que o valor de retorno em Java é declarado antes do nome do método, enquanto na UML ele é definido depois do nome do método e dos parâmetros que ele receberá.

Os atributos de uma classe podem ainda ter características extras, entre as quais podemos citar valores iniciais, multiplicidade e se o atributo é derivado, ou seja, se seus valores são produzidos por meio de algum tipo de cálculo, conforme apresentado na figura 4.3.

Ao estudarmos esse exemplo podemos verificar que o valor inicial dos atributos `situacao` e `saldo` será respectivamente 1 e 0 quando da instanciação de um objeto dessa classe durante a abertura de uma nova conta. Dessa forma, sempre que uma nova conta for aberta sua situação inicial terá o valor 1 (está ativa), e o seu saldo permanecerá com valor 0 até que um depósito seja realizado.

Pode-se perceber também que o atributo `dt_encerramento`, após a definição de seu tipo (`Date`), contém os valores `[0..1]`. Isso é chamado multiplicidade e, nesse contexto, significa que existirá no mínimo nenhuma (0) e no máximo uma (1) data de encerramento para a conta, uma vez que a conta pode estar aberta ainda e, portanto, não poderá ter uma data de encerramento e, se ela tiver sido encerrada, não poderá ter mais do que uma.

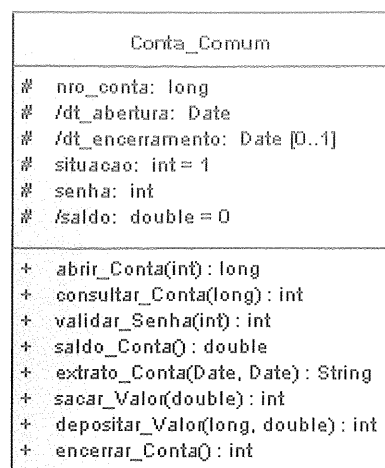


Figura 4.3 – Detalhamento dos Atributos.

Finalmente, os atributos `dt_abertura`, `dt_encerramento` e `saldo` têm uma barra (/) antes de seus nomes, significando que os valores desses atributos sofrem algum tipo de cálculo. No caso das datas, quando for realizada a operação de abertura de conta, o valor da data de abertura será tomado da data do sistema, o mesmo ocorrendo com o valor da data de encerramento quando do encerramento da conta. A rigor, não necessariamente isso poderia ser considerado um cálculo e sim uma simples atribuição, sendo que alguns poderiam considerar isso como sendo o valor inicial dos atributos. No entanto, principalmente no caso da data de encerramento, que será deixada indefinida até que a conta seja encerrada, isso não seria verdadeiro, e o valor desse atributo seria definido em uma operação posterior à criação do objeto. Já no caso do saldo, ele precisa ser recalculado sempre que uma operação de saque ou depósito for realizada.

4.2 Relacionamentos ou Associações

As classes costumam ter relacionamentos entre si, chamados associações, que permitem que elas compartilhem informações entre si e colaborarem para a execução dos processos executados pelo sistema. Uma associação descreve um vínculo que ocorre normalmente entre os objetos de uma ou mais classes.

As associações são representadas por linhas ligando as classes envolvidas. Essas linhas podem ter nomes ou títulos para auxiliar a compreensão do tipo de vínculo estabelecido entre os objetos das classes envolvidas nas associações. Há outras informações que uma associação poderá conter, conforme será visto ao longo do capítulo, na medida em que serão apresentadas as diversas formas de relacionamento possíveis em um diagrama de classes.

4.2.1 Associação Unária ou Reflexiva

Este tipo de associação ocorre quando existe um relacionamento de um objeto de uma classe com objetos da mesma classe. Um exemplo de associação unária pode ser observado na figura 4.4.

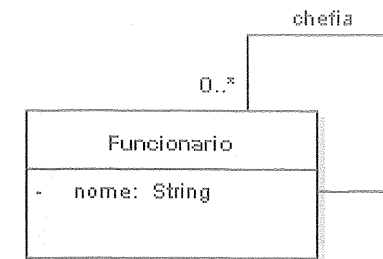


Figura 4.4 – Associação Unária.

Ao examinarmos a figura 4.4, é possível perceber que a única classe do exemplo tem o nome `Funcionario`, e como atributo o nome do funcionário. Observe que uma linha intitulada “chefia” parte da classe `Funcionario` e atinge a própria classe. Isso ocorre porque, nesse exemplo, um funcionário pode ser chefe de outros funcionários. O chefe de um funcionário também é, por sua vez, um funcionário da empresa e, portanto, também se constitui em uma instância da classe `Funcionario`. Logo, a associação denominada “chefia” indica uma possível relação entre uma ou mais instâncias da classe `Funcionario` com outras instâncias da própria classe `Funcionario`, ou seja, essa associação determina se um funcionário pode ou não chefiar outros funcionários.

Observem que existe outra informação na associação, além de seu próprio nome, representada pelo valor “0..*”; Essa informação é conhecida como multiplicidade. A multiplicidade procura determinar o número mínimo e máximo de objetos envolvidos em cada extremidade da associação, além de permitir especificar o nível de dependência de um objeto para com os outros envolvidos na associação.

No caso apresentado na figura 4.3, a multiplicidade 0..* indica que um determinado funcionário pode chefiar nenhum (0) ou muitos (*) funcionários, ou seja, um funcionário pode não chefiar ninguém ou pode chefiar um ou mais funcionários. Pode-se perceber que só existe multiplicidade em uma das extremidades, por default, quando não existe multiplicidade explícita e entende-se que a multiplicidade é “1..1”, significando que um e somente um objeto dessa extremidade da associação relaciona-se com os objetos da outra extremidade. Nesse exemplo, significa que um funcionário pode ou não chefiar outros funcionários, mas um funcionário tem um e apenas um funcionário como chefe.

imediatamente. A tabela 4.1 demonstra alguns dos diversos valores de multiplicidade que podem ser utilizados em uma associação.

Tabela 4.1 – Exemplos de multiplicidade

Multiplicidade	Significado
0..1	No mínimo zero (nenhum) e no máximo um. Indica que os objetos das classes associadas não precisam obrigatoriamente estar relacionados, mas se houver relacionamento indica que apenas uma instância da classe relaciona-se com as instâncias da outra classe (ou da outra extremidade da associação, se esta for unária).
1..1	Um e somente um. Indica que apenas um objeto da classe relaciona-se com os objetos da outra classe.
0..*	No mínimo nenhum e no máximo muitos. Indica que pode ou não haver instâncias da classe participando do relacionamento.
*	Muitos. Indica que muitos objetos da classe estão envolvidos na associação.
1..*	No mínimo um e no máximo muitos. Indica que há pelo menos um objeto envolvido no relacionamento, podendo haver muitos objetos envolvidos.
3..5	No mínimo três e no máximo cinco. Estabelece que existem pelo menos três instâncias envolvidas no relacionamento e que podem ser quatro ou cinco as instâncias envolvidas, mas não mais do que isso.

Outra informação que pode, por vezes, ser considerada útil é a definição de papéis, que são informações extras na associação que podem ajudar a explicar a função de um objeto (o papel que este representa) dentro da associação. Um exemplo do uso de papéis pode ser visto na figura 4.5.

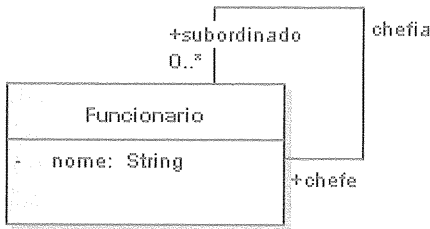


Figura 4.5 – Associação Contendo Papéis.

Nesse exemplo percebemos claramente qual a função de cada objeto envolvido na associação, sendo que o objeto na extremidade cuja multiplicidade é 1 executa o papel de chefe, enquanto os objetos na extremidade de multiplicidade muitos, interpretam o papel de subordinados. Como pode-se perceber, os papéis desse exemplo têm visibilidade pública, mas podem ter visibilidades protegidas ou privadas. O uso de papéis pode facilitar a compreensão da associação existente, mas nem sempre é necessário, e seu uso excessivo pode deixar os diagramas visualmente muito poluídos.

Na verdade, a classe `Funcionario` é uma classe persistente (embora não a tenhamos identificado explicitamente como tal), ou seja, suas instâncias deverão ser preservadas de alguma maneira em um banco de dados. No entanto, não é necessário definir atributos do tipo chave primária ou chave estrangeira, como no modelo entidade-relacionamento. Como regra geral, cada classe, ao ser declarada, já tem um código interno implícito, não sendo necessário declarar um atributo exclusivamente para isso. Ao final do capítulo trataremos da questão de persistência e mapeamento de classes em tabelas em bancos de dados relacionais.

4.2.2 Associação Binária

Associações binárias ocorrem quando são identificados relacionamentos entre objetos de duas classes distintas. Esse tipo de associação é, em geral, a mais comumente encontrada. A figura 4.6 demonstra um exemplo de associação binária.

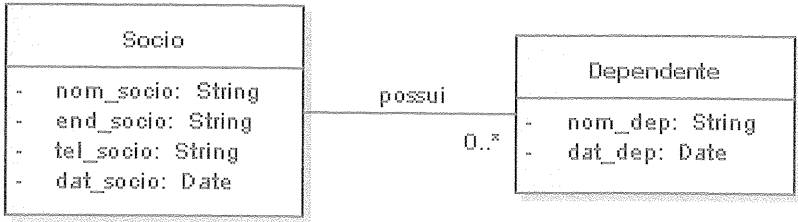


Figura 4.6 – Associação Binária.

Como podemos verificar na figura 4.6, um objeto da classe `Socio` pode relacionar-se ou não com instâncias da classe `Dependente`, conforme demonstra a multiplicidade `0..*`, enquanto se existir um objeto da classe `Dependente` ele terá de se relacionar obrigatoriamente com um objeto da classe `Socio`, pois, como não foi definida a multiplicidade na extremidade da classe `Socio`, isto significa que esta é `1..1`. Poderíamos acrescentar informações a essa associação, como definir a navegabilidade da mesma, conforme demonstrado na figura 4.7.

A navegabilidade é representada por uma seta em uma das extremidades da associação, identificando o sentido em que as informações são transmitidas entre os objetos das classes envolvidas, ou seja, o sentido em que os métodos poderão ser disparados. Nesse exemplo, um objeto da classe `Socio` poderá disparar métodos em objetos da classe `Dependente`, mas a recíproca não é verdadeira: um objeto da classe `Dependente` não poderá disparar métodos em um objeto da classe `Socio`. A navegabilidade não é obrigatória, mesmo porque, se não houver setas, significa que as informações podem trafegar entre os objetos de todas as classes da associação.

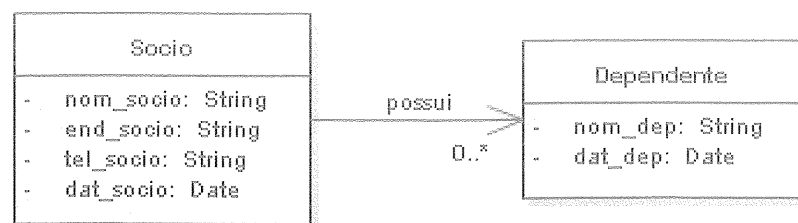


Figura 4.7 – Associação binária com Navegabilidade.

A navegabilidade é uma informação que só deve ser acrescentada na fase de projeto, porém é possível durante a fase de análise definir a direção de leitura da associação, representada, em algumas ferramentas CASE, como um triângulo em forma de seta ao lado do nome da associação, em outras ferramentas, porém, não há diferenciação entre a seta de direção de leitura e a de navegabilidade. A direção de leitura e a navegabilidade não são exatamente a mesma coisa, o sentido de leitura tem como objetivo somente facilitar a compreensão da associação, enquanto a navegabilidade define em que sentido os métodos poderão ser disparados. No exemplo da figura 4.6, o uso da direção de leitura nos transmitiria a informação de que a leitura da associação deveria ser feita da seguinte forma: “Uma instância da classe Socio possui, no mínimo, nenhuma instância e no máximo, muitas instâncias da classe Dependente e uma instância da classe Dependente é possuída por uma e somente uma instância da classe Socio”.

4.2.3 Associação Ternária ou N-ária

Associações ternárias ou n-árias são associações que conectam objetos de mais de duas classes. São representadas por um losango para onde convergem todas as ligações da associação. A figura 4.8 apresenta um exemplo de associação ternária.

Nessa ilustração identificamos uma associação que demonstra um fato corriqueiro na maioria das universidades, em que um professor pode lecionar para muitas turmas, uma turma pode ter muitos professores e utilizar muitas salas de aula e um professor, ao lecionar para uma turma específica, pode utilizar mais de uma sala de aula (ele pode ministrar aulas teóricas em salas comuns ou práticas em laboratório). Assim, podemos ler a associação apresentada na figura 4.8 da seguinte forma: “Um professor leciona para no mínimo uma turma e no máximo para muitas, uma turma tem no mínimo um professor e no máximo muitos, e um professor, ao lecionar para uma determinada turma, utiliza no mínimo uma sala de aula e no máximo muitas”. As associações ternárias são úteis para demonstrar associações complexas. No entanto, deve-se evitar utilizá-las, pois sua leitura é, por vezes, difícil de ser interpretada.

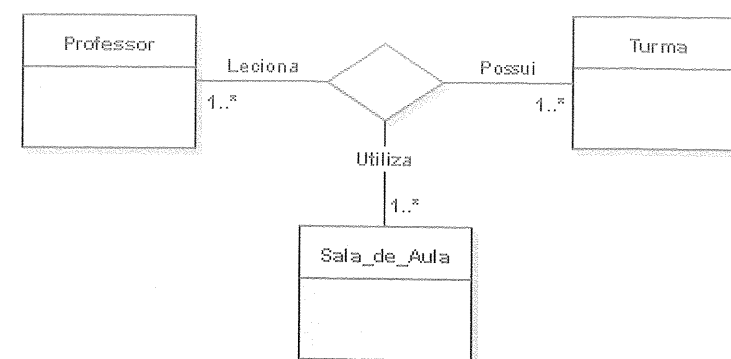


Figura 4.8 – Associação Ternária.

4.2.4 Agregação

Agregação é um tipo especial de associação onde se tenta demonstrar que as informações de um objeto (chamado objeto-todo) precisam ser complementadas pelas informações contidas em um ou mais objetos de outra classe (chamados objetos-parte). Esse tipo de associação tenta demonstrar uma relação todo/parte entre os objetos associados. O símbolo de agregação difere do de associação por conter um losango na extremidade da classe que contém os objetos-todo. A figura 4.9 demonstra um exemplo de agregação.

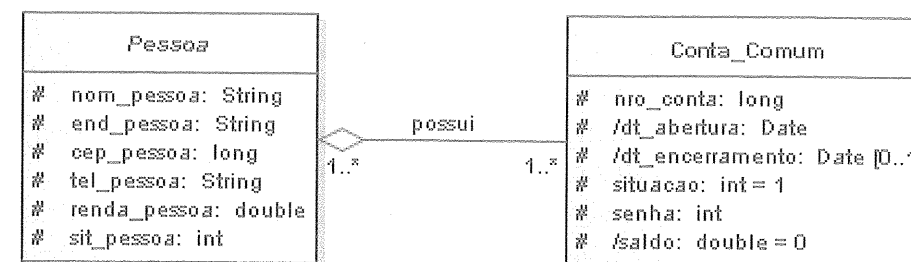


Figura 4.9 – Agregação.

Esse exemplo demonstra uma associação de agregação existente entre uma classe **Pessoa** e uma classe **Conta_Comum**, o que determina que os objetos da classe **Pessoa** são objetos-todo que precisam ter suas informações complementadas pelos objetos da classe **Conta_Comum**, que nessa associação são objetos-parte. Dessa maneira, sempre que uma pessoa for consultada, além de suas informações pessoais, serão apresentadas todas as contas que ela possui.

Ao observarmos as multiplicidades dessa associação perceberemos que, da mesma forma que uma pessoa pode possuir muitas contas, uma conta pode ser possuída por muitas pessoas, como no caso de uma conta conjunta. Isso é característico das agregações nas quais os objetos-parte podem ser compartilhados por mais de um objeto-todo.

A associação de agregação pode, em muitos casos, ser substituída por uma associação binária simples, dependendo da visão de quem faz a modelagem. A função principal de uma associação do tipo agregação é identificar a obrigatoriedade de uma complementação das informações de um objeto-todo por seus objetos-parte, quando este for consultado. Já em uma associação binária essa obrigatoriedade não está explícita.

4.2.5 Composição

Uma associação do tipo composição constitui-se em uma variação da agregação, onde é apresentado um vínculo mais forte entre os objetos-todo e os objetos-parte, procurando demonstrar que os objetos-parte têm de estar associados a um único objeto-todo. Em uma composição os objetos-parte não podem ser destruídos por um objeto diferente do objeto-todo ao qual estão relacionados. O símbolo de composição diferencia-se graficamente do símbolo de agregação por utilizar um losango preenchido. Da mesma forma que na agregação o losango deve ficar ao lado do objeto-todo. A figura 4.10 apresenta um exemplo de composição.

Observando-se a figura 4.10 é possível perceber que um objeto da classe *Revista_Cientifica* refere-se a, no mínimo, um objeto da classe *Edicao*, podendo se referir a muitos objetos dessa classe, e que cada instância da classe *Edicao* relaciona-se única e exclusivamente a uma instância específica da classe *Revista_Cientifica*, não podendo relacionar-se com nenhuma outra.

Ainda nesse exemplo, percebemos que um objeto da classe *Edicao* deve se relacionar a no mínimo seis objetos da classe *Artigo*, podendo se relacionar com até 10 objetos da já citada classe. Esse tipo de informação torna-se útil como documentação e serve como uma forma de validação, que impede que uma revista seja publicada sem ter no mínimo seis artigos, ou mais do que 10. No entanto, um objeto da classe *Artigo* refere-se unicamente a um objeto da classe *Edicao*. Isso é também uma forma de documentação, pois uma edição de uma revista científica só deve publicar trabalhos inéditos. Assim, é lógico que não é possível a um mesmo objeto da classe *Artigo* relacionar-se a mais de um objeto da classe *Edicao*.

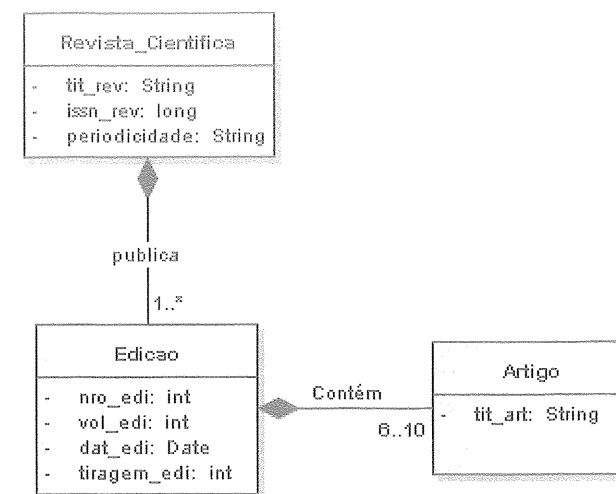


Figura 4.10 – Composição.

4.2.6 Generalização/Especialização

Este é um tipo especial de relacionamento, similar à associação de mesmo nome utilizada no diagrama de casos de uso. O objetivo dessa associação é representar a ocorrência de herança entre as classes, identificando as classes-mãe (ou superclasses), chamadas gerais, e classes-filhas (ou subclasses), chamadas especializadas, demonstrando a hierarquia entre as classes e possivelmente métodos polimórficos nas classes especializadas.

Como no diagrama de casos de uso, a generalização/especialização ocorre quando existem duas ou mais classes com características muito semelhantes. Assim, para evitar ter de declarar atributos e/ou métodos idênticos e como uma forma de reaproveitar código, cria-se uma classe geral em que são declarados os atributos e métodos comuns a todas as classes envolvidas no processo e, então, declaram-se classes especializadas ligadas à classe geral, que herdam todas as suas características, podendo ter atributos e métodos próprios.

Além disso, métodos podem ser redeclarados em uma classe especializada, com o mesmo nome, mas comportando-se de forma diferente, não sendo, portanto, necessário modificar o código-fonte do sistema em relação às chamadas de métodos das classes especializadas, pois o nome do método não mudou, somente foi redeclarado em uma classe especializada, e só se comporta de maneira diferente quando for chamado por objetos dessa classe. A figura 4.11 apresenta um exemplo de generalização/especialização. O símbolo de generalização/especialização é o mesmo do diagrama de casos de uso.

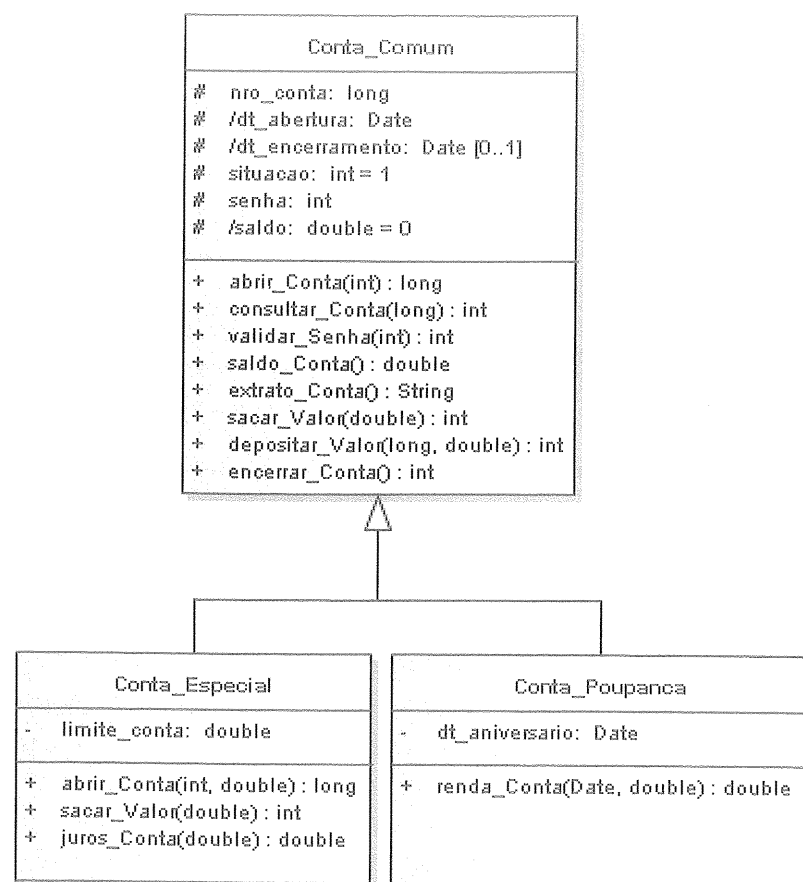


Figura 4.11 – Generalização/Especialização no Diagrama de Classes.

No exemplo da figura 4.11, tomamos a classe *Conta_Comum* já apresentada na figura 4.3, tornando-a uma classe geral (embora ela não seja uma classe abstrata) e derivamos duas classes especializadas a partir dela, as classes *Conta_Especial* e *Conta_Poupanca*, que herdaram suas características.

Além dos atributos e métodos herdados, a classe *Conta_Especial* contém ainda o atributo *limite*, que determina quanto o cliente pode sacar além de seu saldo, e os métodos *abrirConta*, *sacarValor* e *jurosConta*. Os dois primeiros métodos são uma redeclaração dos métodos *abrirConta* e *sacarValor* da classe *Conta_Comum*, pois estes precisam incluir o limite da conta. O último método calcula o valor de juros a ser cobrado pelo uso do *limite*.

Já a classe *Conta_Poupanca* contém, além dos atributos e métodos herdados, o atributo *dt_aniversario*, que define a data em que a conta renderá juros, e o método *rendaConta*, cuja função é calcular os rendimentos a serem acrescidos ao saldo da conta sempre que esta fizer aniversário.

O leitor poderia achar necessário criar uma classe *Conta* que fosse realmente geral, que não tivesse instâncias e servisse apenas para definir os atributos e métodos comuns a todas as contas, ou seja, uma classe abstrata e, a partir desta, derivar as classes *Conta_Comum*, *Conta_Especial* e *Conta_Poupanca*. Isso não estaria errado, apenas não consideramos necessário fazê-lo, por acreditarmos que todas as características de uma conta comum sejam também contidas nas outras classes. Assim, além de definir as características de uma conta comum, a classe pode servir também como classe geral, a partir da qual possam ser derivadas as classes *Conta_Especial* e *Conta_Poupanca*. No caso de realmente ser identificada uma classe abstrata, por convenção, seu nome deve ser escrito em *itálico*.

4.2.7 Classe Associativa

Classes associativas são aquelas produzidas quando da ocorrência de associações que tenham multiplicidade muitos (*) em todas as suas extremidades. As classes associativas são necessárias nos casos em que existem atributos relacionados à associação que não podem ser armazenados por nenhuma das classes envolvidas. A figura 4.12 apresenta um exemplo de classe associativa.

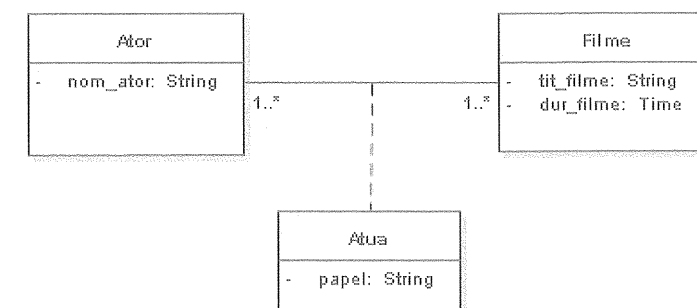


Figura 4.12 – Classe Associativa.

Nesse exemplo, uma instância da classe *Ator* pode se relacionar com muitas instâncias da classe *Filme*, e uma instância da classe *Filme* pode se relacionar com muitas instâncias da classe *Ator*, ou seja, um ator pode atuar em muitos filmes, e um filme pode ter muitos atores atuando nele. Ocorre que existe a necessidade de saber qual o papel interpretado por um ator em um determinado filme, mas onde armazenar essa informação?

Como existe a multiplicidade muitos nas extremidades de ambas as classes da associação, não há como reservar atributos para armazenar as informações decorrentes da associação em quaisquer das classes. Além disso, seria um desperdício de espaço fazê-lo, pois não há como determinar um limite para a quantidade de

atores que poderiam atuar em um filme nem há como saber em quantos filmes um ator poderá atuar. Poderíamos reservar certo número de atributos em uma das classes para armazenar essa informação, mas, na maioria das vezes, diversos deles seriam deixados em branco e, em alguns casos, poderiam não ser suficientes.

Assim, é preciso criar uma classe para guardar essa informação, representada pela classe *Atua*, que conterà o atributo *papel*, referindo-se este exclusivamente a um ator específico atuando em um determinado filme. Uma classe associativa pode perfeitamente ter métodos também. Classes associativas são válidas somente quando existe um único objeto relacionado a duas instâncias associadas, no caso do exemplo, um ator atuando em um filme terá um único papel. Caso um ator interpretasse dois papéis em um mesmo filme, o uso da classe associativa não seria o mais adequado, sendo necessário inserir uma classe normal atuando como classe intermediária da associação, conforme demonstra a figura 4.13.

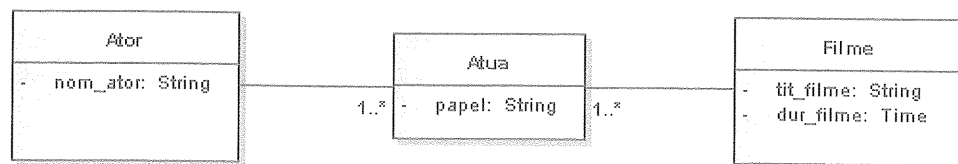


Figura 4.13 – Classe Intermediária.

Na figura 4.13 utilizamos o mesmo exemplo da figura 4.12, substituindo a classe associativa por uma classe intermediária, representando uma situação em que um ator poderia atuar em muitos filmes e um filme poderia ter muitos atores, diferindo-se da situação anterior por permitir que um ator interprete mais de um papel no mesmo filme.

4.2.8 Dependência

Este relacionamento, como o próprio nome diz, identifica certo grau de dependência de uma classe em relação à outra. O relacionamento de dependência é representado por uma linha tracejada entre duas classes, contendo uma seta apontando para a classe da qual a classe posicionada na outra extremidade do relacionamento é dependente.

O relacionamento de dependência é utilizado também em diversos outros diagramas, tendo algumas variações definidas por estereótipos, como include ou extend no diagrama de casos de uso, instantiate no diagrama de objetos ou merge no diagrama de pacotes.

4.2.9 Realização

Uma realização é um tipo de relacionamento especial que mistura características dos relacionamentos de generalização e dependência, sendo usada para identificar classes responsáveis por executar funções para outras classes. Esse tipo de relacionamento herda o comportamento de uma classe, mas não sua estrutura. O relacionamento de realização é representado por uma linha tracejada contendo uma seta vazia que aponta para a classe, que tem uma ou mais funções que devem ser realizadas por outra, enquanto na outra extremidade da linha é definida a classe que realiza esse comportamento. A associação de realização pode ser comparada à palavra-chave *implements* da linguagem Java, que determina que uma classe implementará os métodos definidos em outra classe, ou seja, realizará o comportamento de outra classe. A figura 4.14 apresenta um exemplo de relacionamento de dependência e realização.

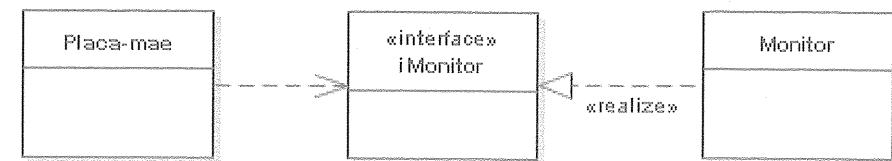


Figura 4.14 – Relacionamento de Dependência e Realização.

Como podemos observar nessa figura, a classe *Placa-Mae* tem um relacionamento de dependência com a classe *iMonitor*, determinando que a classe *Placa-Mae* necessita, de alguma forma, dessa interface. Já a classe *Monitor*, por sua vez, tem um relacionamento de realização com a classe *iMonitor*, o que determina que a classe *Monitor* implementa algum dos serviços oferecidos pela classe *iMonitor*. Observe que a classe *iMonitor* tem um estereótipo *<<interface>>* para destacar sua função. Poderíamos ter colocado o estereótipo *boundary*, que será estudado neste capítulo, mas preferimos utilizar o estereótipo *interface* para deixar mais clara a função da classe.

4.3 Portas

Uma porta é uma característica estrutural de um classificador que especifica uma interação distinta entre o classificador e seu ambiente ou entre o classificador (em termos de seu comportamento) e suas partes internas. Uma porta pode especificar os serviços que um classificador fornece para seu ambiente, bem como os serviços que o classificador espera de seu ambiente. Uma porta representa, portanto, um ponto de comunicação.

Portas e partes internas serão melhor detalhadas e exemplificadas nos capítulos referentes ao diagrama de estrutura composta e ao diagrama de componentes. É necessário, porém, introduzir esse conceito neste momento por este ser necessário ao tópico seguinte. Portas são representadas como pequenos quadrados colocados sobre a borda da classe, conforme mostra a figura 4.15.

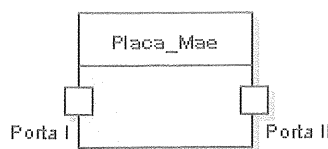


Figura 4.15 – Exemplo de Portas.

Neste exemplo utilizamos novamente a classe Placa-Mae, acrescentando-lhe duas portas, chamadas **Porta I** e **Porta II**, embora não seja obrigatório definir uma nomenclatura para elas. Essa figura servirá de base para os próximos exemplos.

4.4 Interfaces

Atualmente muitas linguagens de programação têm adotado interfaces como um meio para descrever os serviços disponíveis de classes específicas. A partir da versão 2.0 da UML, as interfaces podem ser de dois tipos: fornecidas ou requeridas.

4.4.1 Interfaces Fornecidas

Uma interface fornecida descreve um serviço implementado por uma classe. O conjunto de interfaces implementadas por uma classe forma suas interfaces fornecidas e representa o conjunto de serviços que a classe oferece a seus clientes. Ao implementar uma interface, uma classe suporta o conjunto de características contidas por esta e obedece às suas restrições. As interfaces fornecidas são representadas por um círculo fechado ligado à classe por uma linha sólida. É necessário definir uma porta de comunicação entre a interface e a classe. A figura 4.16 apresenta um exemplo de interface fornecida.

Nesse exemplo continuamos utilizando a classe chamada Placa-Mae, que representa a placa principal de um computador. Essa classe tem uma interface fornecida chamada **iTeclado**, que representa a interface física entre uma placa-mãe e um teclado, onde a placa-mãe (na verdade um de seus componentes, o processador) é responsável por interpretar as teclas pressionadas no teclado.

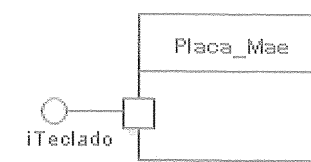


Figura 4.16 – Exemplo de Interface Fornecida.

4.4.2 Interfaces Requeridas

Este tipo de interface descreve os serviços que outras classes devem fornecer a uma determinada classe, que não precisa ter conhecimento de quais classes implementarão esses serviços. As interfaces requeridas são representadas por um semicírculo ligado a uma classe por uma linha sólida. Da mesma forma que nas interfaces fornecidas, é preciso definir uma porta de comunicação entre a interface e a classe. A figura 4.17 apresenta um exemplo de interface requerida.



Figura 4.17 – Exemplo de Interface Requerida.

Aqui utilizamos novamente a classe Placa-Mae, dessa vez contendo uma interface requerida chamada **iMonitor**, que representa a interface física entre uma placa-mãe e um monitor, onde o monitor deve apresentar no vídeo as imagens e os símbolos solicitados pelo processador. É comum que uma interface fornecida em uma classe seja uma interface requerida em outra, podendo facilmente ocorrer de ambas as interfaces surgirem juntas, como demonstra a figura 4.18.



Figura 4.18 – Exemplo de Interface Fornecida e Requerida.

No exemplo da figura 4.18, existem duas interfaces. A primeira representa a interface entre as classes Teclado e Placa-Mae, chamada **iTeclado**, e a segunda representa a interface entre as classes Placa-Mae e Monitor, chamada **iMonitor**. Observe que **iTeclado** é uma interface contida tanto pela classe Teclado quanto pela

Placa_Mae, porém, ela é uma interface requerida para a primeira e uma interface fornecida para a segunda. O mesmo ocorre com a interface iMonitor entre as classes Placa_Mae e Monitor, sendo esta requerida pela primeira e fornecida pela última. Esses exemplos de interfaces serão ainda trabalhados nos capítulos sobre os diagramas de sequência, componentes e estruturas compostas, dentro do escopo de cada diagrama.

É importante notar que as interfaces requeridas e fornecidas podem, às vezes, ser substituídas pelos relacionamentos de dependência e realização já existentes nas versões anteriores, conforme demonstrado na figura 4.14, que apresenta o mesmo exemplo.

4.5 Restrições

Restrições constituem-se em informações extras que definem condições a serem validadas durante a implementação dos métodos de uma classe, das associações entre as classes ou mesmo de seus atributos. As restrições são representadas por textos limitados por chaves. Restrições podem ser usadas para detalhar requisitos não-funcionais, incluindo regras do negócio. A figura 4.19 apresenta um exemplo de restrição.

Nesse exemplo, utiliza-se uma restrição para determinar que um sócio poderá realizar uma locação somente se não tiver nenhuma locação anterior pendente. Esse exemplo enfoca uma restrição que representa uma regra do negócio, ou seja, uma norma que deve ser obedecida quando da execução de um processo. Observe que esta é uma restrição da associação realiza e que está ligada a ela por meio de uma nota. Nesse exemplo, utilizamos uma notação informal e de fácil compreensão, mas existe uma linguagem utilizada para a definição de restrições mais complexas, semelhante a uma linguagem de programação, chamada OCL (Object Constraint Language ou Linguagem de Restrição de Objeto).

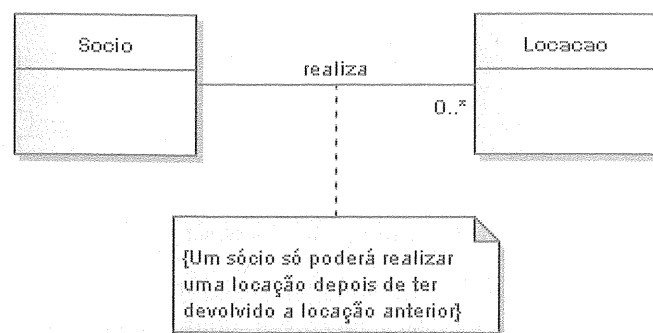


Figura 4.19 – Exemplo de Restrição em uma Associação.

Restrições podem ser aplicadas também para validar um atributo ou método de uma classe específica, como no exemplo apresentado na figura 4.20.

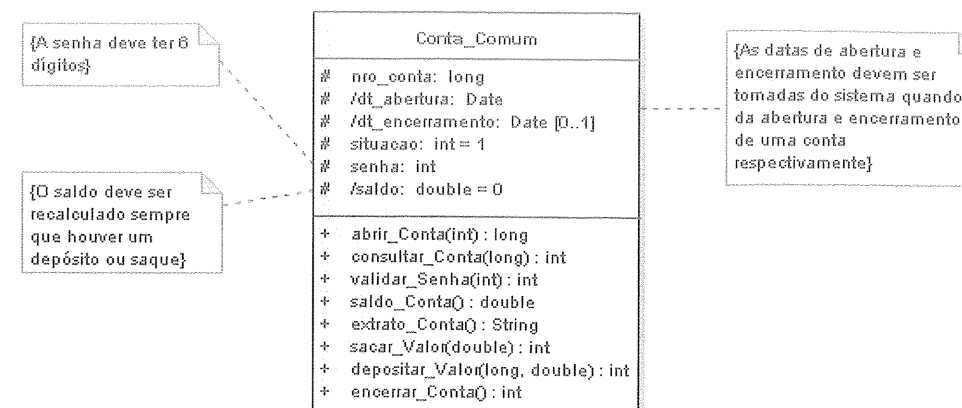


Figura 4.20 – Exemplo de Restrição em Atributos.

No exemplo da figura 4.20, uma nota informa a necessidade de que o atributo senha tenha seis dígitos, restringindo o uso de senhas com um número de dígitos diferentes. Outras notas informam que as datas de abertura e encerramento de conta devem ser tomadas do sistema quando da execução dessas operações e que o saldo da conta deve ser recalculado sempre que um depósito ou saque for realizado. Em alguns casos, é possível encontrar até mesmo detalhes de implementação em uma restrição contida em uma nota, inclusive com instruções escritas em uma linguagem de programação propriamente dita.

Outras tipos de restrições podem ser aplicados a atributos. Por exemplo, pode-se definir um atributo como **{readOnly}** (somente leitura), determinando que seu valor só pode ser lido e não modificado, ou determinar que um atributo é estático (Static). Atributos estáticos são atributos cujos valores são idênticos para todos os objetos de uma classe, ou seja, é um atributo pertencente à classe propriamente dita. Um atributo estático é apresentado sublinhado, como pode ser visto na figura 4.21.

Nesse exemplo, representamos uma classe para um sistema de telefone celular que representa as ligações recebidas ou enviadas pelo aparelho. É preciso registrar todas as ligações recebidas ou feitas, armazenando informações como o número do telefone, o tipo de ligação (se foi feita ou recebida), a data e a hora da ligação, a duração da ligação e se esta foi atendida ou não. Além disso, sempre que uma chamada for recebida, uma música deve tocar. Essa música pode ser escolhida pelo usuário, mas será sempre a mesma para todas as ligações, não havendo uma música

particular para uma ligação específica. Assim, identificou-se o atributo `tom_ligacao` como sendo um atributo estático da classe, que não muda de objeto para objeto.

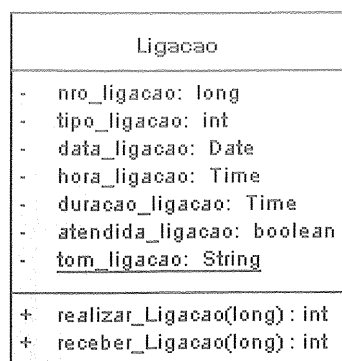


Figura 4.21 – Exemplo de Atributo Estático.

Restrições podem também ser utilizadas para representar o “ou” exclusivo (xor), quando instâncias de duas ou mais classes podem se relacionar com instâncias de outra classe específica, mas somente uma instância de uma das classes pode se relacionar com uma instância da classe em questão, em detrimento das outras. Um exemplo desse tipo de restrição é apresentado na figura 4.22.

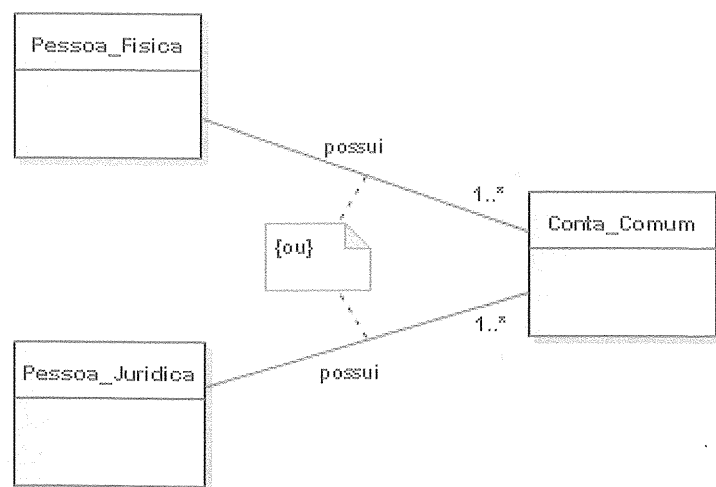


Figura 4.22 – Exemplo de Restrição com ou Exclusivo.

Nesse exemplo, uma conta comum pode pertencer a uma pessoa física ou jurídica, mas uma determinada conta comum só pode pertencer a uma única pessoa. É possível também encontrar classes com restrições abaixo ou ao lado de seus nomes, determinando que a classe tem uma função específica. Embora isso

normalmente seja uma função realizada pelos estereótipos, é possível encontrar classes com restrições abaixo ou ao lado de seus nomes, informando que a classe em questão é uma classe persistente, por exemplo.

Outro exemplo do uso de restrições pode ser aplicado à definição de coleções ordenadas, conforme demonstra a figura 4.23. Uma coleção pode ser definida como um conjunto de instâncias associadas a outro objeto.

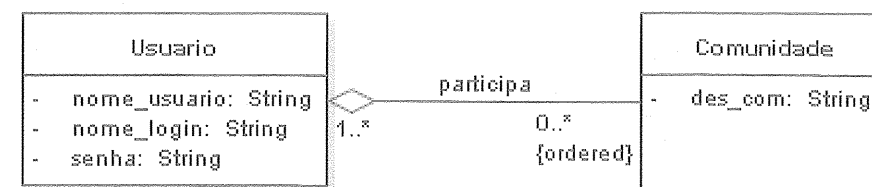


Figura 4.23 – Exemplo de Coleção Ordenada.

Esse exemplo baseia-se em uma rede social semelhante ao Orkut, onde um usuário poderá participar de muitas comunidades. Sempre que um usuário acessar sua conta, será apresentado um determinado número das comunidades de que ele participa. Aqui definimos que o conjunto de comunidades de cada usuário deve ter algum tipo de ordenação (alfabética, por exemplo), conforme demonstra a restrição `{ordered}` abaixo da multiplicidade `0..*` na associação `participa`. Nesse exemplo existe um grande número de comunidades, e um determinado usuário pode participar de um determinado número delas. O conjunto de comunidades associadas a um usuário (não importando que possam estar associadas a outros usuários, como demonstra a associação de agregação) é uma coleção.

Outras restrições que podem ser aplicadas a uma coleção são:

- **unique** – determina que um elemento na coleção não pode se repetir;
- **bag** – permite que um mesmo elemento apareça mais de uma vez;
- **sequence** ou **seq** – representa uma sequência, ou seja, uma **bag** ordenada.

A figura 4.24 apresenta um exemplo de restrição `{unique}` que determina que uma conta corrente não pode se repetir.

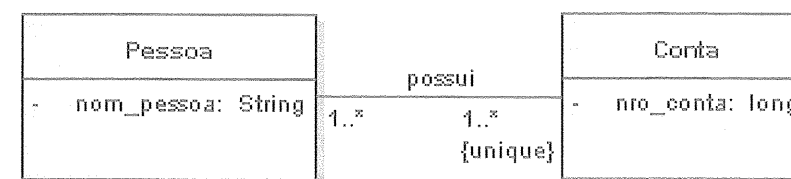


Figura 4.24 – Exemplo de Restrição Unique.

Para que um objeto da conta corrente seja único, é necessário que ao menos um de seus atributos armazene um valor que não se repita em nenhum outro objeto da classe. No caso de uma conta corrente, o número da conta poderia desempenhar essa função, uma vez que nenhum objeto da classê conta poderia armazenar um número de conta igual. Quando existe um atributo único em uma classe é possível criar uma associação qualificada, que é uma forma de identificar individualmente um objeto dentro de uma coleção. O qualificador de uma associação é representado por um pequeno retângulo ligado ao final da associação, conforme demonstra a figura 4.25.

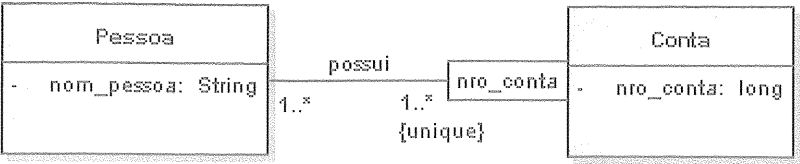


Figura 4.25 – Exemplo de Associação Qualificada.

As restrições podem ainda ser utilizadas para definir melhor a semântica de classes especializadas derivadas de classes gerais. As restrições predefinidas para classes especializadas são:

- **Completa** – quando todas as subclasses possíveis foram derivadas da classe geral.
- **Incompleta** – quando ainda é possível derivar novas subclasses.
- **Separada ou disjunta** – quando as subclasses são mutuamente exclusivas, ou seja, no momento em que uma instância pertence a uma subclasse, não poderá de forma alguma pertencer a qualquer das outras subclasses derivadas. A figura 4.26 fornece um exemplo de classes especializadas separadas e completas.

Nesse exemplo, foram derivadas duas subclasses a partir da classe Pessoa, as classes Fisica e Juridica. Dentro do escopo em que estão inseridas, não existem mais classes que possam ser derivadas a partir da classe Pessoa, por isso, a restrição entre as classes generalizadas é completa. Além disso, no momento em que uma pessoa for física, ela não pode ser jurídica, e vice-versa. Portanto, a especialização também é separada.

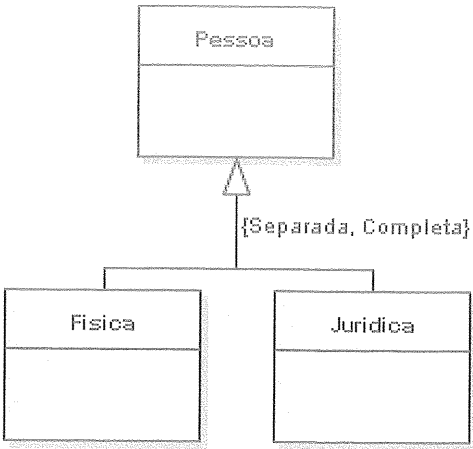


Figura 4.26 – Exemplo de Restrição Separada e Completa.

- **Sobreposta** – quando o fato de pertencer a uma subclasse não impede que pertença a outras. A figura 4.27 apresenta um exemplo de restrição sobreposta e incompleta para classes especializadas.

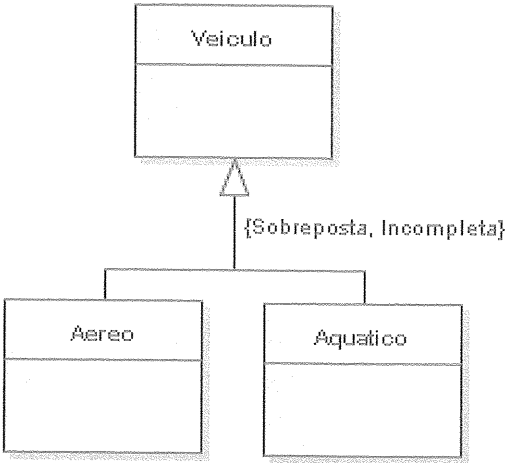


Figura 4.27 – Exemplo de Restrição Sobreposta e Incompleta.

A figura 4.27 apresenta um exemplo no qual, a partir da classe Veiculo, derivaram-se duas subclasses Aereo e Aquatico. Como poderíamos ainda derivar uma classe para veículos terrestres, colocou-se uma restrição incompleta e, como pode ocorrer de um veículo ser tanto aéreo quanto aquático, como é o caso de um hidroavião, acrescentamos ainda a restrição de sobreposta à especialização. Como é possível perceber, as restrições podem ser combinadas, sendo separadas por vírgulas.

Apesar de sua utilidade óbvia, devemos evitar o uso de muitas restrições em um diagrama de classes, porque seu uso excessivo pode tornar o diagrama muito poluído e difícil de ler. Devem-se inserir restrições no diagrama de classes somente quando forem realmente importantes, caso contrário, as restrições devem ser detalhadas em outros diagramas que enfoquem os processos nos quais elas são necessárias.

4.6 Estereótipos do Diagrama de Classes

Os estereótipos foram abordados no final do capítulo 3. Contudo, como já foi dito, eles são uma característica poderosa utilizada em todos os diagramas da UML e, no diagrama de classes, costumam exercer um papel muito importante.

Conforme explanado no capítulo 3, estereótipos são uma maneira de destacar determinados componentes do diagrama, tornando explícito que tais componentes executam alguma função um pouco diferente dos demais componentes apresentados no diagrama. Existem vários estereótipos predefinidos na linguagem UML. Todavia, a linguagem permite a criação de estereótipos particulares por parte de quem a for utilizar.

Há três estereótipos predefinidos na linguagem UML, bastante utilizados nos diagramas de classes que merecem destaque e que serão estudados nas subseções seguintes: os estereótipos `<<entity>>`, `<<boundary>>` e `<<control>>`.

4.6.1 Estereótipo `<<entity>>`

O estereótipo `<<entity>>` tem por objetivo tornar explícito que uma classe é uma entidade, ou seja, a classe contém informações recebidas e armazenada pelo sistema ou geradas por meio deste. Essas informações referem-se ao contexto do problema que o software pretende solucionar. Classes com o estereótipo `<<entity>>` também fornecem a informação de que normalmente terão muitos objetos e que os mesmos possivelmente terão um período de vida longo, isto é, existe a possibilidade de que os objetos dessas classes precisem ser persistidos, ou seja, preservados fisicamente de alguma maneira.

Frequentemente costuma-se confundir classes detentoras do estereótipo `<<entity>>` com classes persistentes. Embora muitas delas realmente o sejam, isso não é uma regra. Uma classe do tipo `<<entity>>` pode ser transiente, isto é, conservar suas informações somente em memória, não sendo necessário preservá-las permanentemente. Quando houver necessidade de declarar de forma explícita que uma classe é persistente, o melhor é criar um estereótipo exclusivo para isso ou colocar uma restrição abaixo ou ao lado do nome da classe

declarando explicitamente que a classe é persistente. A figura 4.28 apresenta um exemplo de classe com o estereótipo `<<entity>>`.

Nesse exemplo, deixamos explícito que a classe `Conta_Comum` é uma entidade, ou seja, que armazena informações referentes ao problema que o sistema no qual ela está inserida procura solucionar, o que não quer dizer que a classe seja persistente. Observe que esse estereótipo é um estereótipo gráfico, ou seja, ele modifica o desenho-padrão da classe. Um dos problemas de se utilizar esse estereótipo é que ele esconde os atributos e métodos da classe. Por outro lado, isso pode ser vantajoso em diagramas grandes, onde pode ser útil não apresentar essas características. A figura 4.29 apresenta a mesma figura com o estereótipo de persistente.

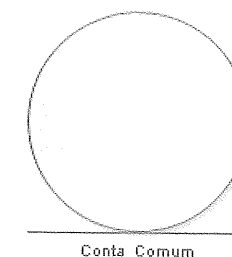


Figura 4.28 – Classe `Conta_Comum` com o estereótipo `<<entity>>`.

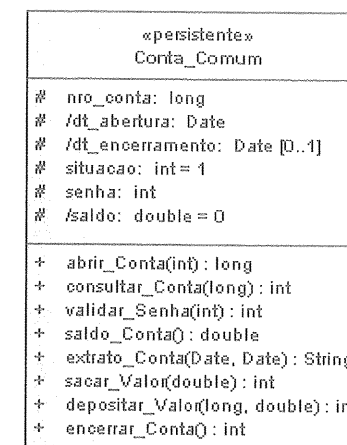


Figura 4.29 – Classe `Conta_Comum` com o estereótipo `<<persistente>>`.

Já nesse exemplo, utilizamos a mesma classe da figura anterior, mas utilizamos um novo estereótipo chamado `<<persistente>>` para deixar bem claro que a classe tem obrigatoriamente de preservar fisicamente, de alguma forma, suas instâncias.

4.6.2 Estereótipo <<boundary>>

O estereótipo <<boundary>>, também conhecido como estereótipo de fronteira, identifica uma classe que serve de comunicação entre os atores externos e o sistema propriamente dito. Muitas vezes uma classe <<boundary>> é associada à própria interface do sistema, embora possa ser definido um estereótipo exclusivo para tanto. Uma classe do tipo <<boundary>> muitas vezes necessita interagir com outra classe do tipo <<control>>, que será explicada na seção seguinte. A utilização de classes <<boundary>> é importante quando é preciso definir a existência de uma interface para o sistema, mas desnecessária em sistemas muito simples, cujas interfaces não apresentam nenhuma característica especial.

4.6.3 Estereótipo <<control>>

O estereótipo <<control>> identifica classes que servem de intermédio entre as classes <<boundary>> e as demais classes do sistema. Os objetos <<control>> são responsáveis por interpretar os eventos ocorridos sobre os objetos <<boundary>>, como os movimentos do mouse ou o pressionamento de um botão, e retransmiti-los aos objetos das classes de entidade que compõem o sistema. A figura 4.30 apresenta um exemplo de classes com estereótipos <<boundary>> e <<control>>.

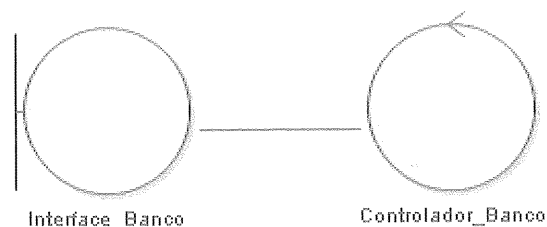


Figura 4.30 – Classes <<boundary>> e <<control>>.

Conforme apresentado na figura 4.30, a classe Interface_Banco representa a interface do sistema e seus objetos serão basicamente rótulos contendo textos, botões e caixas de edição, além do próprio formulário. Os eventos que ocorrem sobre tais objetos são repassados para um objeto da classe Controlador_Banco que interpretará esses eventos e, se necessário, os repassará para os outros objetos que participam do sistema, normalmente na forma de chamada de métodos. Na próxima seção apresentaremos exemplos de diagrama de classes com o uso desses estereótipos.

4.6.4 Estereótipos para Projeto Navegacional

Existem diversos outros tipos de estereótipos com as mais diversas funções, sendo possível utilizar alguns deles para representar o projeto navegacional de um site, por exemplo. Para isso é preciso utilizar estereótipos como <<server page>>, <<client page>> e <<form>>. A figura 4.31 apresenta um exemplo de classe com estereótipo <<server page>>.

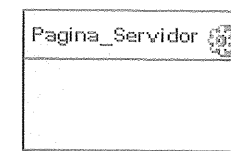


Figura 4.31 – Classe com estereótipo <<server page>>.

O estereótipo <<server page>> é um estereótipo gráfico que apresenta um símbolo de engrenagem ao lado do nome da classe. Esse estereótipo representa uma página web que possui scripts que são executados pelo servidor. A figura 4.32 apresenta um exemplo de estereótipo <<client page>>.

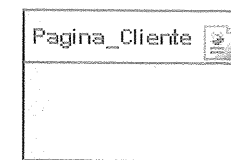


Figura 4.32 – Classe com estereótipo <<client page>>.

O estereótipo <<client page>> representa uma página HTML carregada pelo navegador do usuário, sendo também um estereótipo gráfico apresentando o símbolo de uma página web ao lado do nome da classe. A figura 4.33 apresenta um exemplo de estereótipo <<form>>.

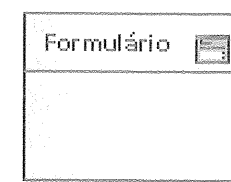


Figura 4.33 – Classe com estereótipo <<form>>.

O estereótipo <<form>> representa um formulário. Em um projeto navegacional representa uma classe que contém um conjunto de campos que fazem

parte de uma página. Este é um estereótipo gráfico que apresenta o desenho de um formulário ao lado da classe. A seguir apresentamos um exemplo de como utilizar esses estereótipos por meio da figura 4.34.

Esse exemplo enfoca uma página de submissões de trabalhos para um evento científico, onde os autores submetem seus trabalhos para avaliação e, se porventura forem aprovados, estes serão publicados nos anais do evento. Dessa forma, existe uma página principal que oferece as alternativas de logar no sistema, submeter um trabalho ou verificar a situação de trabalhos já submetidos.

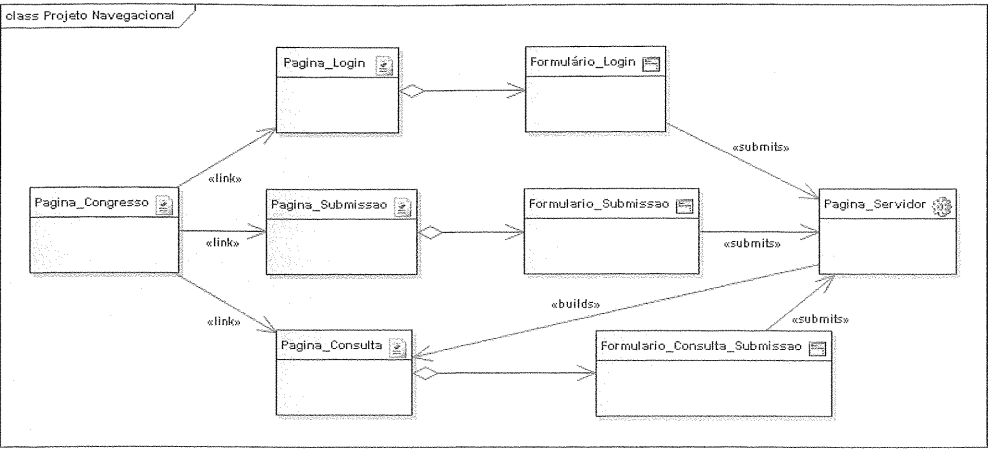


Figura 4.34 – Projeto Navegacional Utilizando Estereótipos.

A página principal é representada pela classe `Pagina_Congresso`, detentora do estereótipo `<<client page>>`, que contém uma associação binária com as classes `Pagina_Login`, `Pagina_Submissao` e `Pagina_Consulta`, que contêm o mesmo estereótipo. Observe que essa associação tem um estereótipo `<<link>>`, representando o vínculo entre as páginas da web que tais classes representam. Uma instância da classe `Pagina_Congresso` nada mais é do que a página propriamente dita carregada na máquina de um usuário.

Cada uma dessas três páginas contém uma associação de agregação com uma classe detentora do estereótipo `<<form>>`, o que determina que a informação da classe `Pagina_Login` precisa ser completada pela informação contida na classe `Formulário_Login`; que a informação da classe `Pagina_Submissao` precisa ser completada pela informação contida na classe `Formulário_Submissao`; e que a informação da classe `Pagina_Consulta` precisa ser complementada pela informação contida na classe `Formulário_Consulta_Submissao`. Essas classes com estereótipo `<<form>>` representam os formulários que deverão ser apresentados quando as páginas representadas pelas classes de estereótipo `<<client page>>` forem carregadas.

Observe que essas três classes com estereótipo `<<form>>` contêm uma associação binária com a classe `Pagina_Servidor`, detentora do estereótipo `<<server page>>`, e que essas associações têm o estereótipo `<<submits>>` indicando que os valores de seus campos serão submetidos à classe `Pagina_Servidor`, que processará essas informações de alguma forma ou as repassará a outras classes, se for necessário. Observe ainda que a classe `Pagina_Servidor` tem também uma associação binária com a classe `Pagina_Consulta`, e essa associação tem o estereótipo `<<builds>>`, significando que a `Pagina_Servidor` pode mandar reconstruir essa página atualizando suas informações. A navegabilidade apresentada no exemplo demonstra o sentido em que trafegam as informações.

Existem vários outros estereótipos que podem ser aplicados a projetos navegacionais, como `<<asp page>>`, `<<jsp page>>`, `<<servlet>>`, `<<web page>>` etc.

4.6.5 Estereótipo <<enumeration>>

Há ainda vários outros estereótipos que podem ser aplicados a uma classe, como, por exemplo, o estereótipo `<<enumeration>>`. Uma enumeração é um tipo de dado cujos valores são enumerados no modelo como literais de enumeração. Basicamente essa classe lista todos os valores válidos que um tipo de dados pode assumir, sendo utilizada principalmente em metamodelos e, embora não costume possuir associações, é geralmente posta próxima das classes que utilizam o tipo de dados cujos literais são por ela enumerados. A figura 4.35 apresenta um exemplo de enumeração que representa os tipos de visibilidade possíveis, enumerados como seus literais.

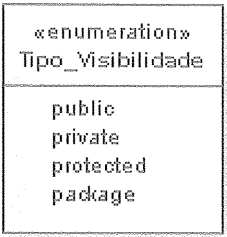


Figura 4.35 – Exemplo de Enumeração.

4.7 Exemplo de Diagrama de Classes – Sistema de Controle Bancário

Nesta seção modelaremos o diagrama de classes – modelo conceitual e de domínio – para o sistema de controle bancário iniciado no capítulo 3, sobre o diagrama de casos de uso. Conforme já foi explicado, o modelo conceitual é produzido

durante a fase de análise e refere-se ao domínio do problema, enquanto o modelo de domínio refere-se ao domínio da solução. O modelo conceitual apresenta somente as informações que o sistema irá necessitar, enquanto o modelo de domínio toma o modelo conceitual e detalha questões como métodos, navegabilidade e até mesmo pode inserir novas classes, se isso for considerado necessário.

Neste exemplo apresentaremos os dois modelos, enquanto nos exemplos seguintes apresentaremos somente o modelo de domínio. É importante destacar que o modelo de domínio somente deve ser desenvolvido na fase de projeto e deve ser modelado junto com o detalhamento dos casos de uso por meio de diagramas de interação, como o de sequência, onde se percebe quais métodos serão necessários em cada processo.

Cumpra ainda destacar que, apesar de haver recursos já descritos como agregação, composição, generalização/especialização ou mesmo classes associativas, não é obrigatória sua utilização em todo diagrama de classes. Esses recursos deverão ser utilizados eventualmente, quando houver necessidade. A figura 4.36 apresenta o modelo conceitual para o sistema de controle bancário.

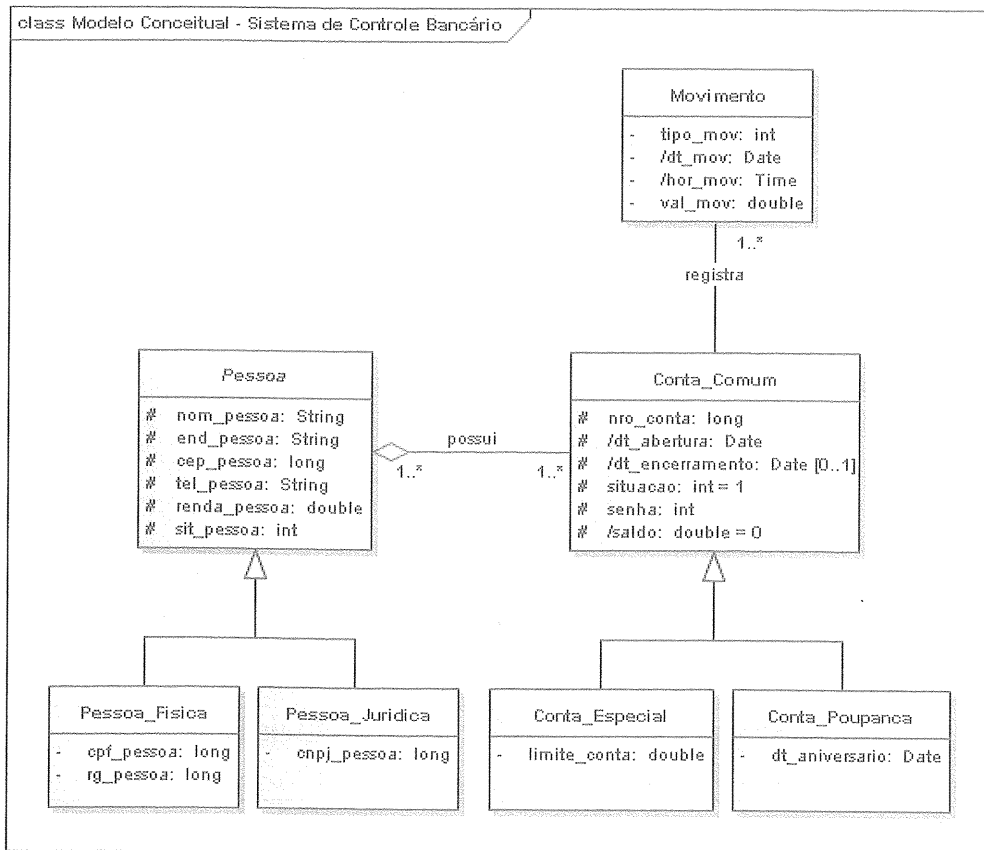


Figura 4.36 – Diagrama de Classes para o Sistema de Controle Bancário – Modelo Conceitual.

Neste modelo foram identificadas as seguintes classes:

1. Conta_Comum

Essa classe já foi parcialmente descrita nas seções anteriores. Seu objetivo é armazenar as contas correntes comuns, sem limite de cheque especial, gerenciadas pelo banco. Nesse tipo de conta não é possível realizar saques além do valor real depositado. A classe Conta_Comum está associada a duas especializações ou subclasses que herdam seus atributos e métodos, representadas pelas classes Conta_Especial e Conta_Poupanca.

A classe Conta_Comum tem os atributos número da conta (nro_conta) do tipo long, data da abertura (dt_abertura) e data de encerramento (dt_encerramento) do tipo Date, situacao (se está ativa ou inativa), senha do tipo int e saldo do tipo double. Observe que os atributos da classe Conta_Comum são todos protegidos, o que os torna visíveis às suas subclasses. Os atributos dessa classe têm ainda características extras, como a definição de que dt_abertura, dt_encerramento e saldo são atributos que receberão algum tipo de cálculo (ou atribuição), que o atributo dt_encerramento não necessariamente precisará ser informado e que os valores iniciais de situacao e saldo serão respectivamente 1 (ao abrir uma conta ela se tornará ativa) e 0 (enquanto nenhum depósito for realizado).

É importante destacar que o tipo Date refere-se a uma classe e que, se esta não for implementada pela linguagem adotada para a codificação do sistema, deverá ser criada junto com as classes aqui definidas.

Nas seções anteriores apresentamos exemplos de restrições que poderiam ser aplicadas aos atributos dessa classe, mas com o objetivo de não deixar o diagrama muito poluído, com uma grande quantidade de informações, preferimos detalhar esse tipo de informação em diagramas separados, quando necessário. Poderíamos acrescentar ainda as restrições readonly aos atributos nro_conta e dt_abertura, uma vez que, após a criação, o número da conta e sua data de abertura não podem ser modificados.

2. Conta_Especial

A classe Conta_Especial representa as contas que permitem ao correntista sacar valores superiores a seu saldo, até um limite estabelecido. Essa classe tem um atributo particular, além dos herdados da classe Conta_Comum, chamado limite_conta do tipo double, que representa o limite máximo que o correntista pode retirar além do saldo de sua conta.

3. Conta_Poupanca

A classe *Conta_Poupanca*, como o próprio nome diz, representa as contas-poupança mantidas pela instituição bancária. Essa classe tem um único atributo particular, além dos herdados, chamado *dt_aniversario* do tipo *Date*, que representa a data em que o valor depositado na conta renderá juros, que não é necessariamente equivalente à data de abertura, uma vez que as contas-poupança não exigem um depósito quando de sua abertura.

4. Pessoa

A classe *Pessoa* armazena as informações gerais dos clientes. Tanto pessoas físicas como jurídicas podem possuir contas na instituição bancária. A classe *Pessoa* tem uma associação do tipo agregação com a classe *Conta_Comum*, o que significa que uma ou mais instâncias dessa classe ou de suas subclasses complementam as informações da classe *Pessoa*, embora possam ser consultadas e manipuladas independentemente. Assim, sempre que uma determinada pessoa for consultada, as informações sobre suas contas devem ser também apresentadas.

Observe que a multiplicidade da associação possui existente entre as classes *Pessoa* e *Conta_Comum* é muitos (*) em ambas as extremidades, significando que uma pessoa pode possuir no mínimo uma e no máximo muitas contas e que uma conta pode ser possuída por uma ou mais pessoas, como no caso de contas conjuntas. O leitor pode se perguntar se não haveria necessidade de inserir uma classe associativa ou intermediária entre essa associação, porém isso somente seria necessário se houvesse um atributo específico relativo a uma determinada pessoa possuindo uma determinada conta que não pudesse ser armazenado nem na classe *Pessoa* nem na classe *Conta_Comum*.

É preciso notar que a classe *Pessoa* é uma superclasse abstrata (e por isso seu nome está em itálico), que não tem instâncias e, portanto, não interage realmente com a classe *Conta_Comum* ou suas especializações. As classes que interagem com a classe *Conta_Comum* são as subclasses *Pessoa_Fisica* e *Pessoa_Juridica*, que herdam todos os atributos da classe *Pessoa* e ainda têm seus próprios atributos particulares.

Os atributos da classe *Pessoa* são nome, endereço e telefone do tipo *String*, CEP do tipo *long*, renda do tipo *double* e *situacao* (se está ativa – há contas ainda não encerradas – ou inativa) do tipo *int*.

5. Pessoa_Fisica

A classe *Pessoa_Fisica* é uma subclasse derivada da classe *Pessoa*, representando as pessoas físicas que possuem ou possuíram contas ativas na instituição bancária. Essa classe herda todos os atributos de sua superclasse, tendo os atributos extras CPF e carteira de identidade (RG) do tipo *long*.

6. Pessoa_Juridica

Essa classe também é uma subclasse derivada da classe *Pessoa*, representando as pessoas jurídicas que possuem ou possuíram contas ativas na instituição. Da mesma maneira, a classe *Pessoa_Juridica* herda todos os atributos da classe *Pessoa*, possuindo somente o atributo particular CNPJ da empresa do tipo *long*.

7. Movimento

Essa classe é responsável por armazenar as transações ocorridas nas contas, ou seja, todos os saques e depósitos. A classe *Movimento* tem como atributos o tipo de movimento (convencionamos atribuir 0 para saque e 1 para depósito) do tipo *int*, a data do movimento (*dt_mov*) do tipo *Date*, a hora do movimento (*hor_mov*) do tipo *Time* e o valor movimentado, do tipo *double*. Observe que o tipo *Time*, da mesma forma que o *Date*, refere-se a classes e não a tipos primitivos, portanto, se essas classes não forem implementadas pela linguagem, terão que ser igualmente codificadas. A classe *Movimento* tem uma associação binária simples com a classe *Conta_Comum*. A multiplicidade dessa associação demonstra que uma conta terá no mínimo um movimento (após a abertura é obrigatório depositar algum valor) e no máximo muitos. Essa associação não caracteriza uma composição porque as informações da classe *Movimento* não complementam as informações da classe *Conta*, quando esta for ser consultada.

A seguir apresentaremos o modelo de domínio desse sistema, onde detalhamos os métodos necessários às classes e acrescentamos classes *boundary* e *control*, como pode ser visto na figura 4.36.

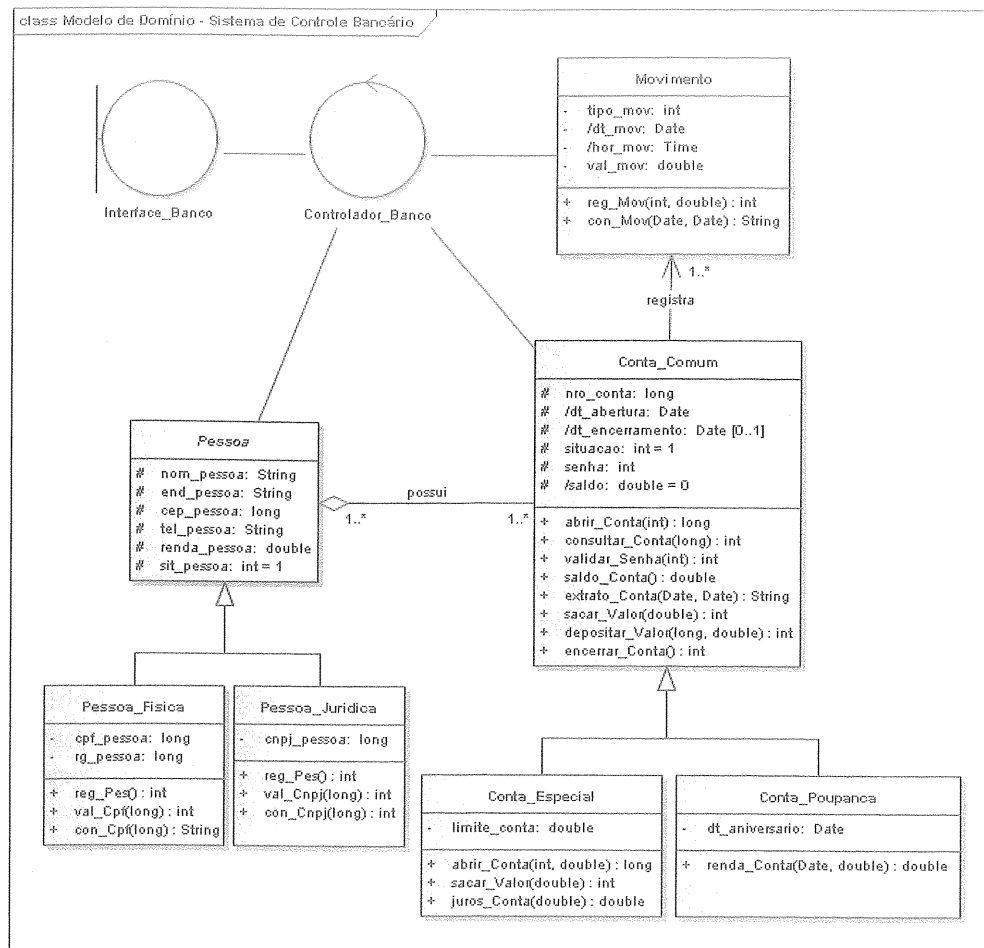


Figura 4.37 – Diagrama de Classes para o Sistema de Controle Bancário – Modelo de Domínio.

A figura 4.37 apresenta o mesmo diagrama de classes para o sistema de controle bancário já explicado anteriormente, mas desta vez o diagrama representa o modelo de domínio e enfoca a solução do problema a ser resolvido pelo sistema. Como podemos observar, foram acrescentados métodos, navegabilidade e duas classes extras:

- **Interface_Banco**, cujo nome é autoexplicativo e representa a interface de comunicação entre os usuários e o sistema, conforme seu estereótipo do tipo **boundary**. Pode ser representada pelos formulários do sistema, manipulados pelos funcionários; pela interface de um caixa eletrônico; ou mesmo pela página do banco, por meio da qual os clientes se autoatendem. As operações realizadas pelos usuários sobre a interface do banco não são tratadas pela classe **Interface_Banco**: ela apenas retransmite os eventos ocorridos em seus objetos para outra classe onde serão interpretados.

- **Controlador_Banco**, que é a classe responsável por interpretar as informações transmitidas pela interface do sistema. Contém o estereótipo **<<control>>**, e sua função é servir de intermediária entre a interface e as classes de entidade que compõem o sistema, solicitando o disparo dos métodos adequados nas instâncias das classes de entidade, de acordo com as opções dos usuários, e retornando à interface os resultados que esta deverá apresentar.

É importante destacar que todas as outras classes do diagrama, com exceção da classe **Pessoa** (que é uma classe abstrata, como demonstra seu nome em *itálico*), são classes de entidade, mas optamos por não utilizar o estereótipo **<<entity>>** porque este, por ser gráfico, modifica o desenho-padrão da classe e esconde seus atributos e métodos, o que atrapalharia a explicação desse diagrama. A seguir detalharemos os métodos identificados para cada classe, que foram descobertos quando do detalhamento dos casos de uso por meio de diagramas de sequência e comunicação que serão explanados nos próximos capítulos.

1. Conta_Comum

Os métodos identificados para essa classe foram:

Método	Descrição
abrir_Conta	A função desse método é abrir uma nova conta, onde um novo objeto dessa classe será instanciado, de forma que esse método agirá como um método construtor. Esse método recebe como parâmetro somente a senha da conta, a ser definida pelo cliente, uma vez que o número da conta é gerado automaticamente e retornado pelo método. A data de abertura é tomada do sistema, a de encerramento é deixada indefinida, e a situação e o saldo têm valores iniciais predefinidos. O retorno do método é o próprio número da conta, se o método for concluído com sucesso, ou zero se ocorrer algum erro durante sua execução.
consultar_Conta	Tem por objetivo descobrir se uma determinada conta existe. Ele recebe um long que armazena o número da conta e retorna um inteiro para determinar se a conta foi encontrada (1) ou não (0).
validar_Senha	Determina se a senha informada é válida. O método recebe a senha como parâmetro e retorna um inteiro determinando se a senha é a correta (1) ou não (0).
saldo_Conta	Retorna o valor contido na conta, retornando um valor double que armazena o saldo da conta. Como este método só pode ser chamado após o método Consulta , não é preciso informar o número da conta, visto que ela já foi informada.

Método	Descrição (cont.)
extrato_Conta	Retorna os movimentos realizados na conta em um determinado período. Recebe como parâmetro as datas iniciais e finais do extrato. O método tem a necessidade de solicitar a execução de outro método sobre a classe Movimento.
sacar_Valor	Diminui o valor solicitado para saque do saldo da conta. Recebe como parâmetro o valor a ser retirado do saldo e retorna verdadeiro se foi possível realizar a operação ou falso, em caso contrário. O retorno falso pode ocorrer devido à conta não ter saldo suficiente para o valor solicitado. Como o método Sacar_Valor só poderá ser chamado após a conta ter sido consultada e sua senha validada, não há necessidade de receber o número da conta, pois este já foi informada no momento da consulta da conta.
depositar_Valor	Soma o valor fornecido pelo cliente ao saldo de uma conta. Ele recebe como parâmetro o número da conta (o depósito pode ser relativo a outra conta) e o valor a ser depositado. Tanto esse método como o Sacar_Valor disparam um método na classe Movimento para registrar o movimento realizado sobre a conta.
encerrar_Conta	Encerra uma conta já existente, tornando-a inativa. Não é um método destrutor, pois apenas altera o valor do atributo <code>si tuacao</code> para 0, indicando que a conta encontra-se inativa. Possivelmente faz o mesmo com o cliente se esta for a única conta por ele possuída. Uma conta não pode ser excluída: é preciso mantê-la para fins de histórico bancário. Esse método não recebe parâmetros, já que para ser encerrada a conta precisa ser primeiro consultada por meio do método Consultar_Conta e, depois disso, o número da conta já estará armazenado na memória. Esse método retorna 1 caso a conta tenha sido encerrada com sucesso ou 0, caso não tenha sido possível encerrá-la.

O leitor talvez venha a se perguntar o porquê de o método `abrir_Conta` ser considerado um método construtor. Na verdade, recomenda-se que os métodos construtores e destrutores não sejam colocados no diagrama de classes porque a forma como são implementados varia muito entre as linguagens de programação. Basicamente um método criador aloca memória para uma instância de uma determinada classe e depois determina valores para seus atributos, enquanto um método destrutor libera a memória utilizada por uma determinada instância. Seguindo esse raciocínio, o método `abrir_Conta` pode perfeitamente implementar a rotina para alocar memória para uma nova instância da classe `Conta_Comum` ou pode chamar um método para isso antes de finalizar a abertura da conta.

2. Conta_Especial

A classe `Conta_Especial` herda todos os métodos da classe `Conta_Comum`, mas tem ainda três métodos particulares, a saber:

Método	Descrição
abrir_Conta	Este é uma redeclaração do método <code>abrir_Conta</code> da classe <code>Conta_Comum</code> . O método precisa ser redeclarado devido à necessidade de se incluir o atributo <code>limite</code> , inexistente na classe <code>Conta_Comum</code> , no momento da abertura da conta. Essa redeclaração do método <code>abrir_Conta</code> caracteriza um exemplo de polimorfismo, já que a classe redeclarada tem o mesmo nome, mas tem algumas diferenças em sua implementação.
sacar_Valor	O segundo método da classe <code>Conta_Especial</code> também é uma redeclaração do mesmo método existente na classe <code>Conta_Comum</code> . Esse método precisa ser redeclarado para incluir o uso do atributo <code>limite</code> , pois um correntista que possua uma conta especial pode sacar valores superiores a seu saldo real até o limite estabelecido pelo atributo de mesmo nome. Tal redeclaração caracteriza outro exemplo de polimorfismo, uma vez que a chamada do método permanece a mesma, diferenciando-se na forma como o método é implementado.
juros_Conta	O último método diminui diariamente o valor de juros a ser pago pelo uso do limite do cheque especial do saldo do correntista até que o limite seja coberto. Como podemos perceber, esse método recebe como parâmetro o valor de juros a ser debitado do saldo do cliente. Tal método não recebe um número de conta específico, pois, ao ser chamado, ele verificará todas as instâncias da classe <code>Conta_Especial</code> à procura de contas que estejam utilizando seu limite.

3. Conta_Poupanca

Da mesma forma que a classe `Conta_Especial`, essa classe herda todos os métodos da classe `Conta_Comum`, tendo somente um método extra:

Método	Descrição
renda_Conta	Recebe como parâmetros a data atual e o percentual de juros que as contas com aniversário no dia receberão. O método aplica-se a todas as instâncias cujo dia da data de aniversário seja igual ao dia da data atual.

4. Pessoa_Fisica

Essa classe tem os seguintes métodos:

Método	Descrição
reg_Pes	É responsável por instanciar um novo objeto da classe. Uma vez que esse método recebe como parâmetro todos os atributos da classe, com exceção da situação, cujo valor inicial é 1 (ativo), optamos por não detalhar a assinatura do método para não deixar a classe larga demais, o que atrapalharia a visualização da figura.

Método	Descrição (cont.)
val_CPF	É utilizado para determinar se o CPF informado é válido. Esse método é chamado quando do registro de uma nova pessoa física e recebe como parâmetro um long que armazenará o valor do CPF a validar.
con_Cpf	Permite consultar uma pessoa por seu CPF. Se o CPF for encontrado, ele retornará uma String com os dados do cliente.

5. Pessoa_Juridica

Os métodos dessa classe são descritos a seguir:

Método	Descrição
reg_Pes	Tem a mesma função do método de mesmo nome explicado na classe anterior, diferenciando-se por não registrar o CPF ou o RG da pessoa e sim o CNPJ da empresa.
val_Cnpj	É utilizado para determinar se o CNPJ informado é válido. Esse método é chamado quando do registro de uma nova pessoa jurídica e recebe como parâmetro um long que armazenará o valor do CNPJ a validar.
con_Cnpj	Permite consultar uma pessoa jurídica por seu CNPJ. Se o CNPJ for encontrado ele retornará 1, caso contrário, 0.

6. Movimento

Essa classe tem os métodos explanados a seguir:

Método	Descrição
reg_Mov	É responsável por registrar cada movimento ocorrido em alguma conta. Esse método recebe como parâmetros o tipo de movimento (se foi saque ou depósito), do tipo int e o valor do movimento do tipo double, pois a data e a hora do movimento são capturados diretamente do sistema. O método retornará 1 se conseguir registrar o movimento, caso contrário, 0. Esse método é chamado pelos métodos Sacar_valor e Depositar_valor declarados na classe Conta_Comum.
con_Mov	É utilizado para consultar todos os movimentos dentro de um período, recebe como parâmetro a data inicial e final da consulta e retorna uma string contendo os valores dos atributos de cada objeto.

Aqui é necessário fazer uma observação, principalmente em relação aos exemplos seguintes. A rigor, uma classe deve ter um get e um set para cada um de seus atributos. No entanto, se torna inviável e enfadonho criar esses métodos para cada classe e, principalmente se a classe tiver muitos atributos, tornará a classe muito grande e, por conseguinte, ficará difícil ler a representação do diagrama como um todo. Assim optamos por utilizar métodos genéricos como registrar ou consultar, para instanciar ou retornar todos os valores de uma classe com o objetivo de simplificar o modelo, o que não impede que o leitor proceda de forma diferente, se assim achar necessário.

4.8 Exemplo de Diagrama de Classes – Sistema de Videolocadora

Nesta seção apresentaremos o modelo de domínio representado pelo diagrama de classes do sistema de videolocadora que iniciamos a modelar no capítulo sobre o diagrama de casos de uso. A figura 4.38 apresenta o modelo de domínio referente a esse sistema.

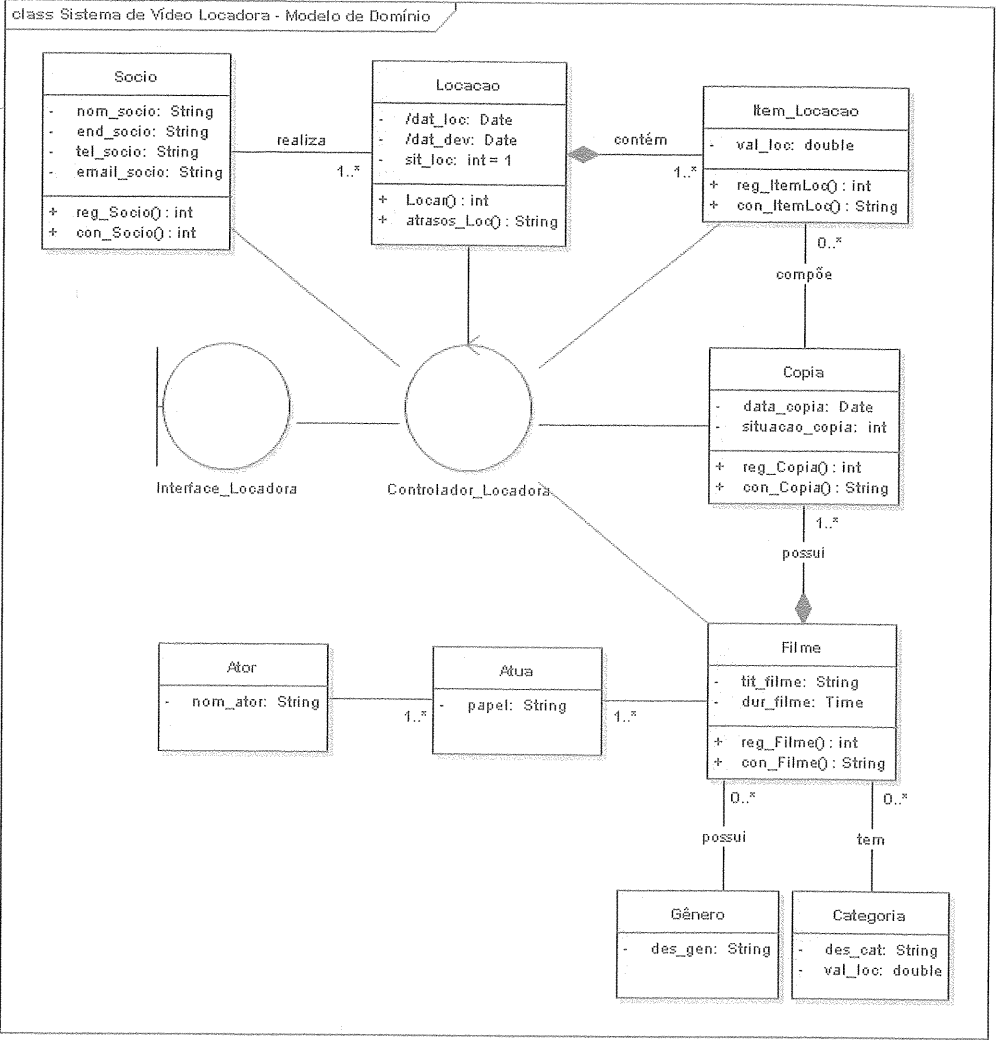


Figura 4.38 – Diagrama de Classes para o Sistema de Videolocadora – Modelo de Domínio.

Ao examinarmos esse diagrama percebemos que existe uma classe de fronteira que representa a interface do sistema, chamada Interface_Locadora e uma classe de controle chamada Controlador_Locadora, responsável por interpretar os eventos da interface e disparar os métodos adequados nas classes de entidade.

As classes controladoras, quando não existe mais de uma responsável por um determinado número de classes, ligam-se em geral, senão a todas, à maioria das classes de entidade, porém, para evitar produzir diagramas com muitas linhas ligando as diversas classes, costuma-se associar a classe controladora só às classes mais importantes. Detalharemos a seguir as classes de entidade identificadas nesse modelo. Algumas delas, como o leitor observará, não têm métodos por trataram-se de classes simples, cuja a única função é a de armazenamento de informação, de modo que não consideramos necessário detalhar seus métodos.

1. Categoria

Essa classe tem como função armazenar as possíveis categorias de filmes oferecidos pela locadora, como, por exemplo, lançamentos, clássicos etc. A categoria influencia no valor de locação. Essa classe tem os atributos de descrição da categoria do tipo `String` e valor de locação do tipo `double`.

2. Gênero

Essa classe tem como função armazenar os possíveis gêneros de filmes oferecidos pela locadora, como aventura, ficção científica, época, romance, terror etc. Essa classe tem como atributo a descrição do gênero do tipo `String`.

3. Ator

Essa classe tem como função armazenar os atores principais que atuem nos filmes oferecidos pela locadora. Isso pode ser útil para fins de pesquisa de filmes por um determinado ator. A classe tem como atributo o nome do ator do tipo `String`.

4. Atua

Esta é uma classe intermediária localizada entre as classes `Ator` e `Filme` e armazena os papéis que um determinado ator interpreta em cada filme em que atuou. Seu único atributo é o papel interpretado do tipo `String`. Observe que a classe tem multiplicidade muitos nas duas associações com a classe `Ator` e com a classe `Filme`, determinando que um ator pode interpretar muitos papéis em muitos filmes e que um filme pode ter no mínimo um ator e no máximo muitos atores atuando nele.

5. Filme

Essa classe armazena as informações dos filmes que a locadora possui cópias. Esta classe tem como atributos o título do tipo `String` e a duração do filme do tipo `Time`. A classe tem também o método `reg_Filme`, utilizado para registrar os filmes adquiridos pela empresa e o método `con_Filme`, utilizado para consultar filmes já registrados. Essa classe tem associações binárias com as classes `Categoria`, `Gênero` e `Atua`, uma vez que um filme deve ter uma categoria e um gênero, e no mínimo um ator deve atuar nele. Observe que, apesar de um filme ter uma única categoria, uma categoria pode pertencer a muitos filmes, o mesmo ocorrendo com o gênero.

6. Cópia

Essa classe tem por função armazenar as informações referentes às cópias de cada filme pertencentes à locadora. Observe que a locadora não pode emprestar o filme em si, apenas suas cópias. A classe `Filme` armazena as informações dos filmes, mas a classe `Cópia` refere-se à cópia física existente na locadora para empréstimo. Os atributos relevantes para essa classe são a data em que a cópia foi adquirida (`data_copia`), do tipo `Date`, e a situação da cópia (se está disponível, emprestada ou inadequada para locação), do tipo `int`. A classe tem ainda os métodos `reg_Cópia`, responsável por registrar as novas cópias ou atualizar sua situação, e `con_Cópia`, responsável por consultar as cópias existentes.

Observe que a classe `Filme` tem uma associação de composição com a classe `Cópia`, o que determina que um filme pode ter no mínimo uma e no máximo muitas cópias e que os objetos da classe `Cópia` são objetos-parte que complementam a informação dos objetos-todo da classe `Filme` e, além disso, que um objeto `Cópia` só pode estar relacionado a um objeto `Filme`.

7. Socio

Essa classe armazena as informações dos sócios que podem locar filmes na locadora. Seus atributos são nome, endereço, telefone e e-mail, todos do tipo `String`. A classe contém também os métodos `reg_Socio` e `con_Socio`, que permitem o registro de novos sócios e a consulta de sócios já existentes, respectivamente.

8. Locacao

O objetivo dessa classe é armazenar as informações referentes à cada locação realizada na locadora. Seus atributos são data de locação e de devolução (dat_loc e dat_dev) do tipo Date, além da situação da locação (locada e devolvida) do tipo int, cujo valor inicial é 1, indicando que a locação ainda não foi devolvida. Se o valor do atributo for 2, a locação já foi devolvida.

Essa classe tem os métodos Locar e atrasos_Loc. O primeiro é responsável por instanciar um novo objeto da classe Locacao, enquanto o segundo deve verificar todas as locações em atraso, ou seja, todas aquelas cuja data de devolução já tenha expirado e sua situação ainda seja 1. Como pode-se perceber, existe uma associação binária entre Socio e Locacao, isto significa que um sócio pode realizar no mínimo uma e no máximo muitas locações e que uma locação deve ser feita por somente um sócio.

9. Item_Locacao

Os objetos da classe Item_Locacao têm a função de complementar as informações dos objetos da classe Locacao, ou seja, são objetos-parte, enquanto os objetos Locacao são objetos-todo. Os objetos dessa classe armazenam informações referentes a cada cópia de filme locada em uma locação. Uma vez que o valor de uma cópia pode aumentar ou diminuir, é preciso guardar o valor da locação na época, sendo esta a função do atributo val_loc do tipo double contido nessa classe.

Observe que uma locação deve ter no mínimo um item de locação e, como demonstra a associação de composição entre as classes Locacao e Item_Locacao, um objeto Item_Locacao deve estar associado exclusivamente a um objeto Locacao. Além disso, existe uma associação binária entre a classe Copia e a Item_Locacao, já que uma cópia pode ser locada muitas vezes, ou seja, compor um item de locação muitas vezes. Assim, um objeto da classe Item_Locacao está associado a somente um objeto da classe Copia, mas um objeto Copia pode estar associado a muitos objetos Item_Locacao.

A classe Item_Locacao contém ainda dois métodos, reg_ItemLoc e con_ItemLoc, que têm a função de registrar um novo item de locação e consultar um item de locação, respectivamente.

4.9 Exemplo de Diagrama de Classes – Sistema de Telefone Celular

Nesta seção apresentaremos o diagrama de classes referente ao modelo de domínio para o sistema de telefone celular, cuja modelagem foi iniciada no capítulo sobre o diagrama de casos de uso. A figura 4.39 apresenta o modelo de domínio desse sistema.

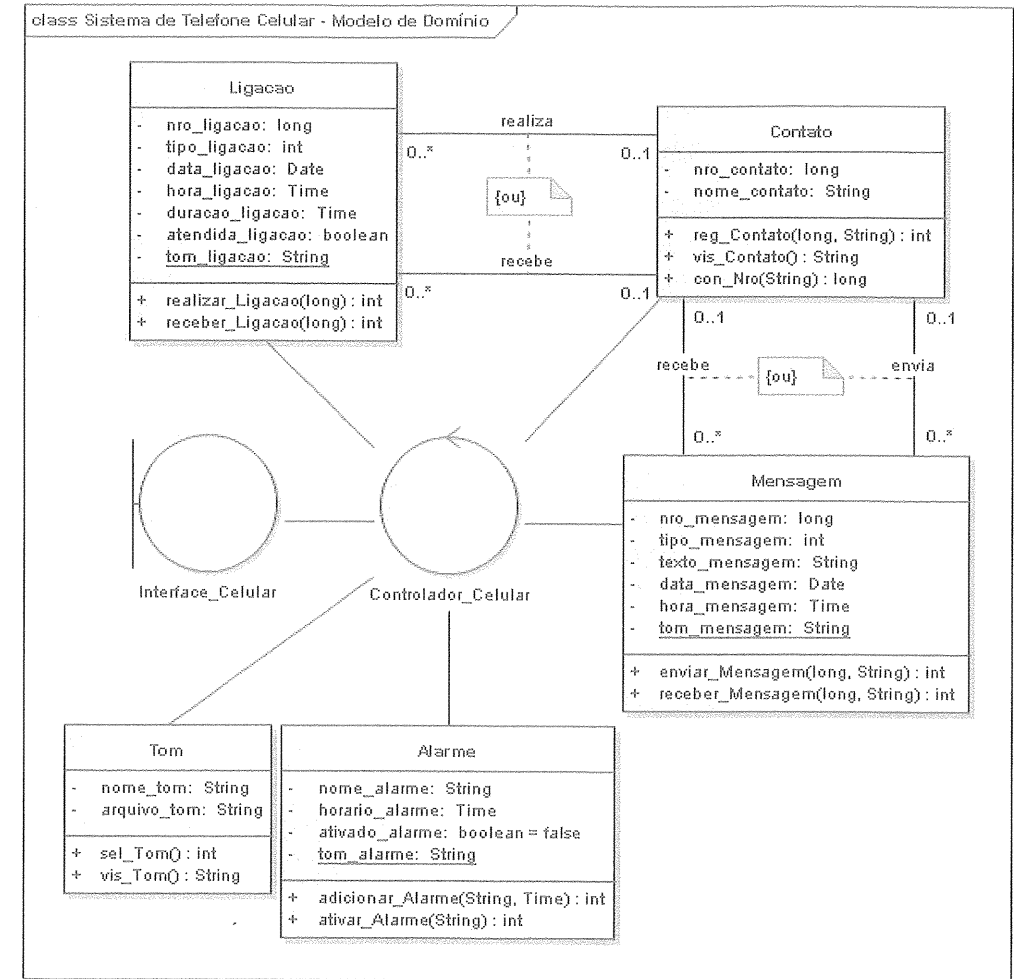


Figura 4.39 – Diagrama de Classes para o Sistema de Telefone Celular – Modelo de Domínio.

Como podemos perceber, o diagrama apresenta a interface do celular representada pela classe Interface_Celular, bem como uma classe responsável por interpretar os eventos ocorridos na interface, chamada Controlador_Celular. A seguir descreveremos as classes de entidade que compõem esse diagrama, junto com seus atributos, métodos e associações.

O leitor notará que incluímos as assinaturas dos métodos das classes, uma vez que o diagrama não é muito grande e o número de parâmetros dos métodos é pequeno, o que permite este detalhamento.

1. Contato

Essa classe é autoexplicativa, sendo seu objetivo armazenar as informações dos contatos que o usuário mantém em seu celular. A classe tem os atributos número do contato do tipo `long` e nome do contato do tipo `String`. A classe apresenta ainda os seguintes métodos:

Método	Descrição
<code>reg_Contato</code>	Tem como função instanciar um novo objeto da classe <code>Contato</code> . Ele retorna um inteiro que representará verdadeiro (1), se a operação for realizada com sucesso, ou falso (0), em caso contrário.
<code>vis_Contato</code>	Executa a tarefa de visualizar os contatos armazenados no aparelho. O método é chamado enquanto houver contatos, apresentando o nome de cada contato na tela do aparelho, para que o usuário possa visualizá-los.
<code>con_Nro</code>	Tem por objetivo consultar um contato a partir da listagem fornecida após a execução do método <code>vis_Contato</code> . O método receberá como parâmetro o nome do contato a consultar e retornará o número correspondente.

2. Ligacao

Essa classe armazena as ligações realizadas e recebidas pelo usuário. Tem como atributos o número da ligação do tipo `long`, o tipo da ligação (se foi realizada ou recebida) do tipo `int`, a data da ligação do tipo `Date`, a hora e a duração da ligação do tipo `Time` e se a ligação foi atendida (verdadeiro) ou não (falso), do tipo `boolean`. A classe contém ainda os seguintes métodos:

Método	Descrição
<code>realizar_Ligacao</code>	Tem a função de fazer uma chamada para outro telefone, recebendo como parâmetro o número a discar e retornando verdadeiro, se a ligação foi realizada, ou falso, em caso contrário. O método deve ainda instanciar um objeto da classe <code>Ligacao</code> , sendo que o tipo da ligação será 1, que por convenção identifica uma chamada enviada. A data e a hora serão tomadas do sistema, a duração será contada enquanto a ligação durar, e o atributo <code>atendida_Ligacao</code> receberá verdadeiro, se a ligação for atendida, ou falso, em caso contrário.

Método	Descrição (cont.)
<code>receber_Ligacao</code>	Executa a tarefa de receber uma ligação, recebendo como parâmetro o número do aparelho que está solicitando atendimento. Esse método também deverá instanciar um objeto da classe <code>Ligacao</code> , sendo a ligação atendida ou não. Se o usuário atender a chamada, o atributo <code>atendida_Ligacao</code> receberá o valor verdadeiro, caso contrário, falso. O valor dos outros atributos será idêntico ao método anterior, excetuando-se o tipo, que receberá o valor 2.

Observe que a classe `Ligacao` tem duas associações binárias com a classe `Contato`. A primeira associação, cujo nome é `realiza`, indica uma situação em que um contato previamente cadastrado liga para o aparelho. Note que um contato pode não ligar nunca para o usuário, ou pode fazer muitas ligações, e que uma ligação pode não ser realizada por nenhum contato (o número pode ser desconhecido) ou somente por um contato específico. A segunda associação refere-se a uma situação em que, a partir da consulta de um contato, o usuário do aparelho faz uma chamada para este, passando como parâmetro o número consultado. Da mesma maneira que na associação anterior, um contato pode receber muitas ligações ou nunca receber nenhuma, e uma ligação pode não ser recebida por nenhum contato cadastrado ou por somente um deles. Observe ainda que existe uma restrição entre as duas associações, do tipo `ou exclusivo`, ou seja, um contato não pode fazer e receber uma mesma ligação.

3. Mensagem

Essa classe armazena as mensagens enviadas e recebidas pelo usuário. Tem como atributos o número do aparelho para o qual a mensagem foi enviada, do tipo `long`; o tipo da mensagem (se foi enviada ou recebida), do tipo `int`; o texto da mensagem, do tipo `String`; a data da mensagem, do tipo `Date`; e a hora da mensagem, do tipo `Time`. Essa classe contém ainda os métodos `enviar_Mensagem` e `receber_Mensagem`, bastante semelhantes aos métodos declarados na classe `Ligacao`, diferenciando-se principalmente por, além do parâmetro do número do telefone, receberem ainda a mensagem a enviar ou receber.

Observe que a classe `Mensagem` tem duas associações com a classe `Contato`, que representam as situações em que um contato envia ou recebe uma mensagem, podendo acontecer de a mensagem não ser enviada ou recebida de nenhum contato conhecido ou estar associada a um único contato. Além disso, uma mensagem, caso esteja associada a um contato, não pode ser enviada e recebida ao mesmo tempo pelo contato, como demonstra a restrição do tipo `ou exclusivo`.

4. Alarme

Essa classe armazena os horários em que o usuário deseja ser despertado. Tem como atributos o nome do alarme, do tipo String; o horário que o alarme deverá despertar, do tipo Time; e o atributo ativado_alarme, que determina se o alarme em questão está ativado (deve despertar) ou não, do tipo boolean. O valor inicial para esse último atributo é falso. A classe contém os seguintes métodos:

Método	Descrição
adicionar_Alarme	Instancia um novo objeto da classe Alarme, recebendo como parâmetros o nome do alarme e seu horário, sendo que o atributo ativado_alarme será iniciado com falso e não precisa ser passado como parâmetro. Esse método retorna verdadeiro, se a operação for realizada com sucesso, e falso, em caso contrário.
ativar_Alarme	Recebe o nome do alarme a ativar (ou desativar, caso já esteja ativado) como parâmetro, modificando o valor do atributo ativado_alarme para verdadeiro, se estivesse falso, ou falso, em caso contrário.

Pode-se notar que as classes Ligacao, Mensagem e Alarme têm um atributo muito semelhante que representa o tom a ser tocado quando uma ligação ou mensagem for recebida ou quando chegar a hora do alarme despertar. Deve-se observar também que este é um atributo estático, ou seja, um atributo que não pertence aos objetos, mas à classe propriamente dita, uma vez que sempre que uma ligação ou mensagem for recebida a mesma música deverá tocar. O atributo tom é uma String porque se refere ao nome do tom que deverá ser tocado, sendo que o aparelho tem um conjunto de tons que podem ser selecionados, mas somente um para as chamadas recebidas, outro para as mensagens recebidas e outro para o despertar do alarme.

5. Tom

Essa classe armazena os tons disponíveis no aparelho. Seus atributos são o nome do tom e o nome do arquivo que o celular deverá procurar para executar a música, ambos do tipo String. Os métodos dessa classe são sel_Tom, que apresenta ao usuário uma lista onde este deverá escolher entre ligação, mensagem ou alarme. Se o proprietário do aparelho escolher uma das opções, o método chamará o método vis_Tom para visualizar os tons disponíveis e permitir que o usuário selecione um deles para a função previamente escolhida. Optamos por não associar essa classe às classes Ligacao, Mensagem ou Alarme por seus objetos não se relacionarem a um objeto específico dessas classes, e sim à classe propriamente dita.

4.10 Exemplo de Diagrama de Classes – Sistema de Clínica Veterinária

Nesta seção apresentaremos o diagrama de classes referente ao modelo de domínio para o sistema de clínica veterinária iniciado no capítulo sobre o diagrama de casos de uso. A figura 4.40 apresenta o modelo de domínio desse sistema.

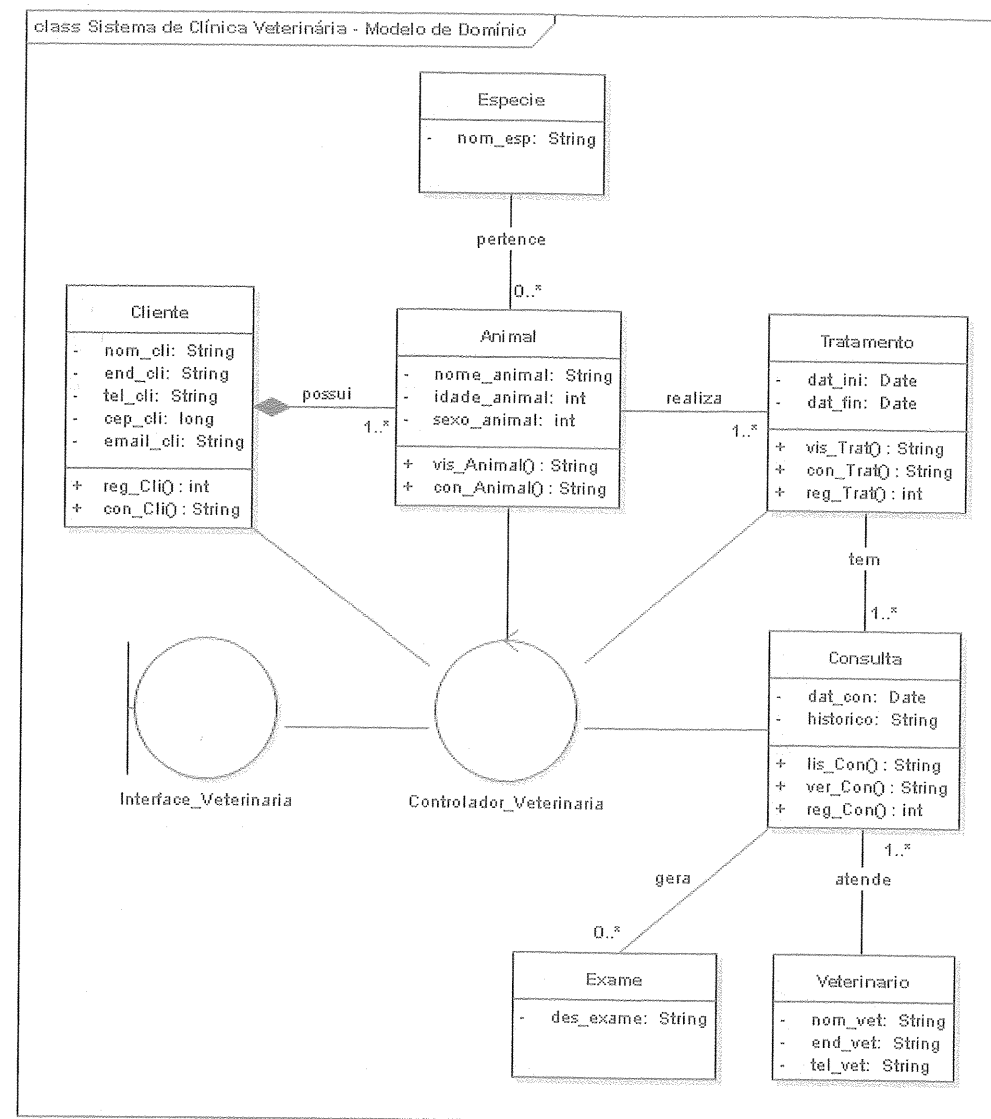


Figura 4.40 – Diagrama de Classes para o Sistema de Clínica Veterinária – Modelo de Domínio.

Como nos exemplos anteriores, incluímos uma classe de interface e outra controladora. Detalharemos a seguir as classes de entidade identificadas nesse modelo. O leitor notará que declaramos somente os métodos considerados importantes nesse modelo.

1. Espécie

Essa classe representa as diversas espécies de animais tratadas na clínica e tem como atributo o nome da espécie do tipo `String`. Uma instância da classe `Especie` pode estar associada a muitas instâncias da classe `Animal`, porém, pode acontecer de nenhuma instância da classe `Animal` relacionar-se com uma determinada instância da classe `Especie`, conforme demonstra a associação denominada `pertence` entre elas.

2. Cliente

Classe de fácil compreensão, cujo objetivo é representar os clientes da clínica. Seus atributos são autoexplicativos. O método `con_Cli` tem como objetivo consultar as informações de um cliente. Já o método `reg_Cli` tem a função de registrar um novo cliente. Observe que um cliente precisa ter suas informações complementadas pelos animais que ele possui, como demonstra a associação de composição entre as classes `Cliente` e `Animal`.

3. Animal

Essa classe também é fácil de compreender, sendo responsável por armazenar as informações dos animais já tratados na clínica. Da mesma forma que na classe `Cliente`, seus atributos são autoexplicativos. O método `vis_Animal` retorna todos os animais que um determinado cliente possui, e o método `con_Animal` permite consultar um dos animais listados. Conforme é possível notar ao examinarmos as associações dessa classe, um animal deve pertencer a um único dono e a uma única espécie, embora um cliente possa ter muitos animais e uma espécie possa referir-se a muitos animais. Além disso, um animal, para estar registrado na clínica, deve ter iniciado ao menos um tratamento, podendo estar associado a muitos deles.

4. Tratamento

Essa classe representa os diversos tratamentos pelos quais passa ou passou um determinado animal. Para estar registrado nesse sistema, uma instância da classe `Animal` deve estar associada à, no mínimo, uma instância da classe `Tratamento`. Um tratamento tem como atributos as datas de seu início e término do tipo `Date`. O método `vis_Trat` permite visualizar todos os tratamentos já realizados por um determinado animal, o método `con_Trat` permite consultar um determinado tratamento, e o método `reg_Trat`, abrir um novo tratamento.

5. Consulta

Essa classe representa cada uma das consultas pelas quais passa um animal durante um tratamento. Um tratamento associa-se a no mínimo uma consulta. Cada instância da classe `Consulta` tem como atributos a data em que a consulta foi realizada, do tipo `Date`, e sua documentação histórica, do tipo `String`. Os métodos dessa classe são semelhantes aos da anterior, em que o método `lis_Con` permite listar todas as consultas de um determinado tratamento, o método `ver_Con` permite consultar uma consulta específica, e o método `reg_Con`, marcar uma nova consulta.

6. Veterinario

As instâncias dessa classe armazenam as informações referentes a cada um dos veterinários que trabalham na clínica. Observe que uma instância da classe `Veterinario` pode se relacionar a muitas instâncias da classe `Consulta`, mas uma instância da classe `Consulta` associa-se a somente uma instância da classe `Veterinario`. Os atributos dessa classe são autoexplicativos.

7. Exame

As instâncias dessa classe armazenam os possíveis exames marcados em uma determinada consulta. Por esse motivo, uma instância da classe `Exame` estará sempre associada a uma instância da classe `Consulta`, porém, uma instância da classe `Consulta` pode se associar a nenhuma (caso o veterinário não marque nenhum exame) ou a muitas instâncias da classe `Exame`. O atributo desta classe armazena a descrição de cada exame solicitado.

4.11 Exemplo de Diagrama de Classes – Sistema de Controle de Advocacia

Nessa seção apresentaremos o diagrama de classes referente ao modelo de domínio para o sistema de controle de advocacia que foi iniciado no capítulo sobre o diagrama de casos de uso. A figura 4.41 apresenta o modelo de domínio desse sistema.

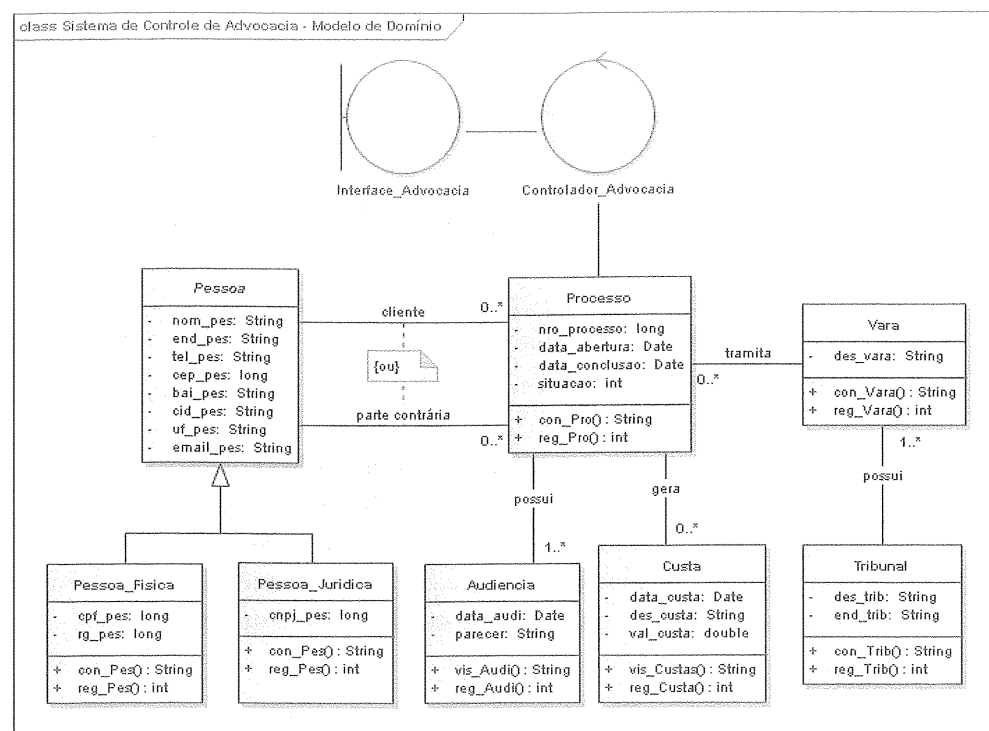


Figura 4.41 – Diagrama de Classes para o Sistema de Controle de Advocacia – Modelo de Domínio.

Como nos exemplos anteriores, incluímos uma classe de interface e outra controladora. Detalharemos a seguir as classes de entidade identificadas nesse modelo.

1. Pessoa

Essa classe representa as pessoas que alguma vez participaram de algum processo como clientes ou partes contrárias. Observe que esta classe é uma classe abstrata, conforme demonstra o texto em itálico de seu nome, cujo único objetivo é armazenar os atributos comuns às duas classes que são especializadas a partir da mesma, as classes *Pessoa_Fisica* e *Pessoa_Juridica*, indicando que um cliente ou parte contrária pode tanto constituir-se em uma pessoa física quanto em uma jurídica. Observemos que tanto uma instância da classe *Pessoa_Fisica* quanto da *Pessoa_Juridica* só pode associar-se como cliente ou parte contrária de uma instância da classe *Processo*, conforme demonstra a restrição “ou” entre as associações. Além disso, uma instância de qualquer dessas duas classes pode nunca ter se associado como cliente ou parte contrária, conforme demonstra a multiplicidade dessas associações. Os atributos dessa classe são autoexplicativos.

2. Pessoa_Fisica

Essa classe armazena os atributos e métodos particulares das pessoas físicas que são clientes ou estão sendo processadas pelo escritório. Seus atributos são o CPF e a carteira de identidade da pessoa, ambos do tipo *long*, e seus métodos *con_Pes* e *reg_Pes*, cujo objetivo é consultar as informações de uma pessoa e registrar uma nova pessoa física, respectivamente.

3. Pessoa_Juridica

Essa classe armazena os atributos e métodos particulares das pessoas jurídicas que são clientes ou estão sendo processadas pelo escritório. Seu atributo é o CNPJ da pessoa, do tipo *long*, e seus métodos *con_Pes* e *reg_Pes*, cujo objetivo é consultar as informações de uma pessoa e registrar uma nova pessoa física, respectivamente. Optamos por declarar esses dois métodos em ambas as classes devido à consulta ser realizada por atributos diferentes (CPF ou CNPJ) e por terem de registrar igualmente diferentes atributos particulares.

4. Processo

Esta é a principal classe desse diagrama e representa todos os processos já concluídos ou em andamento do escritório. Como o número do processo (do tipo *long*) é um valor informado externamente, precisa ser declarado explicitamente na classe. Além deste, os atributos dessa classe armazenam a data em que o processo foi aberto, a data de seu possível término, ambos do tipo *Date*, e a situação do processo, ou seja, se ele se encontra em andamento ou já foi encerrado, do tipo *int*. A classe *Processo* contém ainda os métodos:

Método	Descrição
<i>reg_Pro</i>	Permite a geração de um novo processo.
<i>con_Pro</i>	Permite a consulta de um processo específico. Esse método recebe como parâmetro o número do processo que se deseja consultar e retorna uma <i>String</i> com os dados do processo, se este tiver sido encontrado.

5. Audencia

Essa classe armazena todas as audiências por que passa um determinado processo. Uma instância da classe *Processo* deve estar associada a, no mínimo, uma instância da classe *Audencia*. No entanto, uma instância da classe *Audencia* só pode se associar a uma única instância da classe *Processo*. Os atributos dessa classe contêm a data em que ocorreu a audiência e o parecer do tribunal. Essa classe tem ainda dois métodos:

Método	Descrição
vis_Audi	É utilizado para retornar todas as audiências de um determinado processo.
reg_Audi	Permite o registro de uma nova audiência.

Poderíamos pensar em transformar a associação binária simples entre as classes *Audiencia* e *Processo* em uma composição, mas não consideramos necessário, ao consultar um processo, que todas as suas audiências sejam também apresentadas obrigatoriamente.

6. Custas

Essa classe representa as possíveis custas de um processo. Cada instância dessa classe deve se relacionar com uma instância da classe *Processo*, mas uma instância da classe *Processo* pode se associar com nenhuma ou muitas instâncias da classe *Custas*. Seus atributos são a data em que a custa foi contraída, do tipo *Date*, a descrição da custa, do tipo *String*, e o valor da custa do tipo *double*. A classe tem ainda os métodos descritos a seguir:

Método	Descrição
vis_Custas	Tem por função retornar todas as custas de um determinado processo.
reg_Custas	Permite a geração de uma nova custa.

7. Vara

Um processo tramita em uma determinada vara, mas uma vara pode ter muitos processos em tramitação, conforme demonstra a multiplicidade da associação entre essas classes. O atributo dessa classe armazena a descrição da vara do tipo *String*. Essa classe tem ainda os métodos *con_Vara*, que serve para consultar uma vara específica, e *reg_Vara*, para instanciar um novo objeto da classe.

8. Tribunal

Finalmente, essa classe armazena os tribunais aos quais as varas onde tramitam os processos estão vinculadas. Uma vara pertence a um único tribunal, mas a um tribunal podem pertencer muitas varas, como demonstra a multiplicidade da associação entre essas classes. A classe *Tribunal* tem como atributos o nome e o endereço do tribunal, do tipo *String*, e os métodos *con_Trib*, para consultar um determinado tribunal, e *reg_Trib*, para registrar um novo tribunal.

4.12 Persistência e Mapeamento de Classes em Tabelas

Em muitos casos pode ser necessário preservar de maneira permanente os objetos de uma classe, ou seja, esta deve ser persistente. Uma classe persistente apresenta muitas semelhanças com uma entidade como as definidas no antigo modelo entidade-relacionamento utilizado para definir as estruturas de tabelas em bancos de dados relacionais.

Na verdade, o diagrama de classes foi intencionalmente projetado para ser uma evolução do modelo entidade-relacionamento e pode ser utilizado para modelar a estrutura lógica das tabelas que irão compor um banco de dados. Nas entidades que compunham o modelo entidade-relacionamento, no entanto, eram definidos apenas os dados a serem preservados, os quais são representados nas classes pelos seus atributos. Entretanto, o diagrama de classes oferece ainda a possibilidade de definir as operações que podem ser aplicadas às tabelas, representadas por seus métodos.

Deve ficar claro, no entanto, que nem toda classe é persistente, sendo muitas vezes desnecessário preservar suas informações, quando são chamadas transientes. Desse modo, é preciso identificar quais classes, se existirem, são persistentes, o que é feito por meio do emprego de estereótipos ou de restrições, como detalhado no decorrer deste capítulo. Ao identificar uma classe como persistente, o modelador estará deixando explícito que é preciso preservar em disco, de algum modo, as instâncias da classe.

A forma como as instâncias serão preservadas irá depender da forma como o sistema for implementado, podendo estas ser atualizadas em disco, à medida que o sistema for sendo manipulado, ou mantidas em memória e registradas no disco apenas quando do encerramento do sistema ou da chamada de um método específico para isso. Muitos autores recomendam o uso de uma camada de persistência, ou seja, a derivação de uma subclasse a partir da classe cujas informações necessitam ser mantidas, quando seria definida a maneira como as instâncias de cada classe seriam preservadas.

Quando se utiliza um SGBD orientado a objetos não é necessário preocupar-se tanto com o mapeamento desses objetos. Porém, quando se utiliza um SGBD relacional, é necessário mapear as classes cujos objetos deseja-se persistir em tabelas. Nesses casos é necessário definir quais classes são persistentes e quais são transientes. Classes de fronteira e de controle normalmente são transientes, enquanto classes de entidade normalmente são persistentes. É preciso também definir se todos os atributos de uma classe deverão ser persistidos, o que pode ser feito por meio de restrições aplicadas particularmente aos atributos.

Muitas vezes uma classe persistente terá seu equivalente em forma de tabela. Nesses casos, cada atributo será uma coluna na tabela, e cada objeto se tornará uma linha. Porém, isso não é uma regra: uma classe pode ser dividida em mais de uma tabela ou uma tabela pode representar muitas classes.

Atualmente existem diversos frameworks de persistência que se responsabilizam por persistir os objetos das classes. Entre eles pode-se citar, por exemplo, o Hibernate, o Torque ou o Castor, entre outros. Entretanto, pode ser necessário saber quais tabelas correspondem a quais classes, de modo que descreveremos como mapear classes em tabelas relacionais nas próximas seções.

4.12.1 Estereótipo Table

Nos casos mais simples, uma classe persistente será representada por uma tabela. Nesses casos será apenas preciso definir uma chave primária para a tabela em questão, podendo-se criar um atributo exclusivamente para isso ou utilizar um atributo já existente. A figura 4.42 apresenta o mapeamento de uma classe denominada Socio em uma tabela.

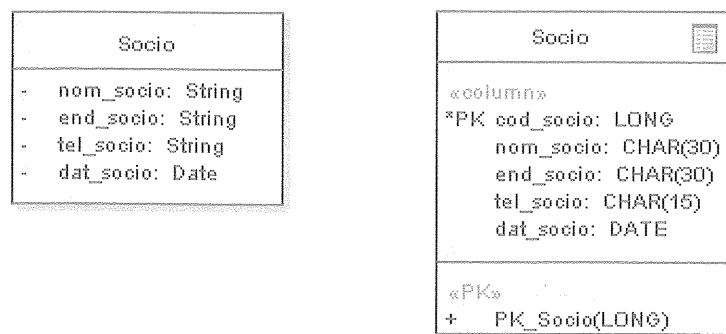


Figura 4.42 – Mapeamento de uma Classe em Tabela.

Nesse exemplo, a classe Socio está representada à esquerda na figura, enquanto a tabela em que ela foi mapeada está à direita. Como podemos perceber, a tabela Socio é uma classe contendo um estereótipo gráfico. O perfil de modelagem de dados representa uma tabela como uma classe estereotipada, ou seja, uma classe com o estereótipo <<table>>.

Ao examinarmos a tabela mapeada, percebemos que suas colunas são quase idênticas aos atributos da classe Socio, acrescentando-se somente uma coluna extra, denominada cod_socio, do tipo LONG, para servir de chave primária, conforme demonstra o texto PK (Primary Key, ou Chave Primária) ao lado da coluna. Uma chave primária é composta por uma ou mais colunas de uma tabela cujo valor ou valores servem para identificar uma linha específica da tabela em questão.

Foi necessário incluir uma nova coluna para servir de chave primária porque nenhum dos atributos da classe Socio poderia assumir essa função, já que uma chave primária precisa ser única, ou seja, seu valor não pode se repetir em nenhuma linha. Observe que a tabela contém uma divisão contendo o estereótipo <<column>> para identificar as colunas da tabela e outra divisão contendo o estereótipo <<PK>> para identificar a chave primária.

4.12.2 Associações e Chaves Estrangeiras

Associação de 1 para 1

Embora isso não seja muito comum, pode acontecer de um objeto em uma das extremidades de uma associação referir-se a somente um objeto da outra extremidade, e esse objeto, por sua vez, também estar associado unicamente ao mesmo objeto da outra extremidade, ou seja, o objeto obj1 está associado somente ao objeto obj2, e o obj2 está associado somente ao obj1.

Muitas vezes, uma associação de 1 para 1 denota um erro de modelagem, o que normalmente demonstra que uma das classes envolvidas na associação não deveria existir e que seus atributos deveriam ser transferidos para a outra classe da associação, deixando a própria associação de existir também. Porém, esse tipo de associação pode ocorrer, como é demonstrado na figura 4.43.

Neste exemplo identificamos uma situação em que existe uma classe Pessoa, e esta tem uma associação unária denominada conjuge, com multiplicidade 1 em seus dois extremos. Percebemos ainda que um objeto dessa classe envolvido na associação interpreta o papel de marido e deve estar associado a um único objeto da mesma classe interpretando o papel de esposa. Essa associação pode ser lida da seguinte forma: um marido é cônjuge de uma e somente uma esposa, e uma esposa é cônjuge de um e somente um marido.

Quando existe uma associação de 1 para 1, deve-se adicionar uma chave estrangeira em uma das tabelas envolvidas na associação para referenciar a chave primária da tabela localizada na outra extremidade da associação. Uma chave estrangeira é uma coluna ou conjunto de colunas que armazena um valor através do qual podemos identificar de forma única uma linha em uma tabela associada, por meio da comparação do valor da chave estrangeira com o valor da chave primária da tabela pesquisada. Nesse caso específico, a chave estrangeira identificará uma linha da própria tabela Pessoa, por isso, essa coluna deverá armazenar o valor de um código de pessoa.

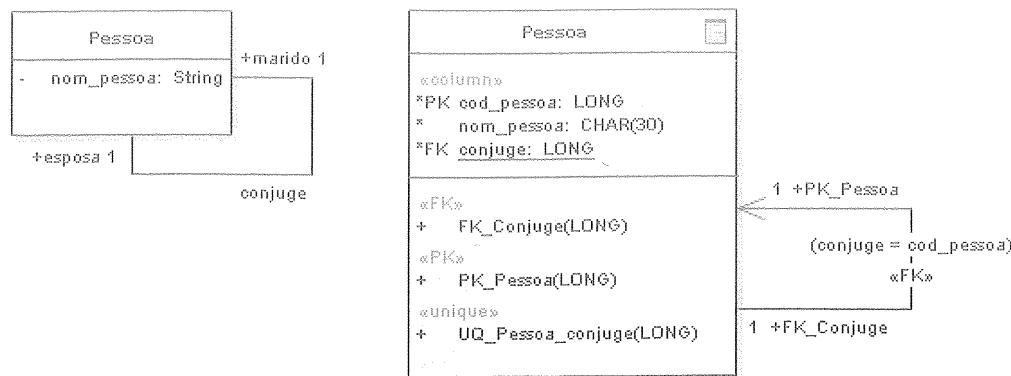


Figura 4.43 – Mapeamento de uma Classe com Associação 1 para 1.

Ao examinarmos a tabela Pessoa perceberemos que foi incluída uma coluna denominada *cod_pessoa*, para servir de chave primária, e que foi criada outra coluna chamada *conjuge* para servir como chave estrangeira, conforme demonstra o texto FK (Foreign Key, ou Chave Estrangeira) ao lado do nome da coluna. Além disso, como esse campo é único (seu valor não pode se repetir em outra linha), ele é apresentado sublinhado. Observe que na terceira divisão da tabela, além da definição da chave primária por meio do estereótipo <<PK>>, foi definida uma chave estrangeira chamada *FK_Conjuge* e detalhou-se quais colunas são únicas, como demonstra o estereótipo <<unique>>. Observe ainda que existe uma associação unária na tabela Pessoa com associação 1 para 1, onde, para que se possa identificar o cônjuge de uma pessoa, é necessário que o valor da chave estrangeira seja igual ao da chave primária.

Associação de 1 para Muitos

Quando existe uma associação de 1 para muitos (*) entre uma ou mais tabelas, deve-se adicionar uma chave estrangeira na extremidade muitos (*) da associação para se referenciar à chave primária da tabela da outra extremidade. Um exemplo disso é apresentado na figura 4.44.

Neste exemplo identificamos uma situação em que existe uma associação binária simples entre as classes Socio e Dependente. Essa associação já foi explicada no início do capítulo. As classes Socio e Dependente foram mapeadas em tabelas equivalentes, sendo acrescentada em cada uma delas uma coluna para identificar a chave primária da tabela. Observe que na tabela Dependente acrescentou-se ainda uma coluna para servir de chave estrangeira, cuja função é identificar um sócio específico. Note que essa coluna não é única, porque mais de um dependente pode estar associado a um mesmo sócio.

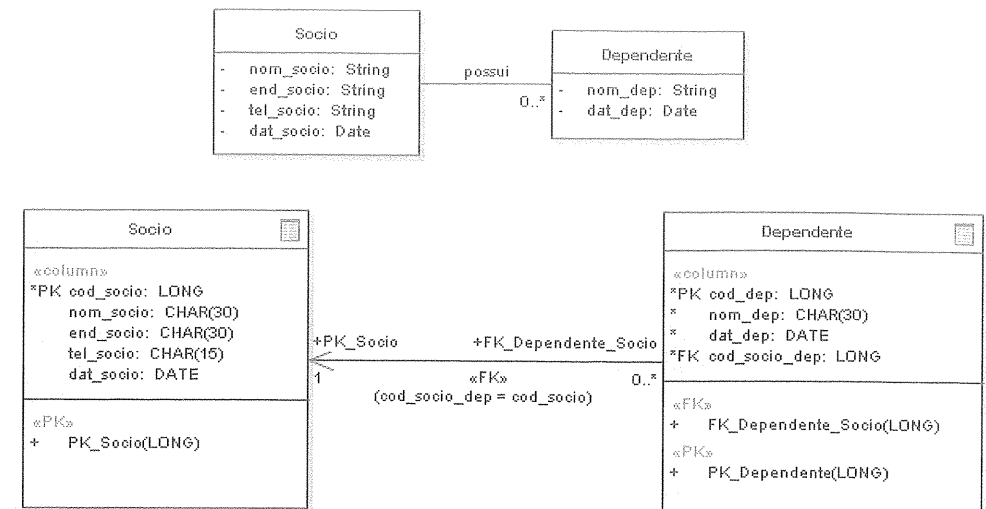


Figura 4.44 – Mapeamento de Classes com Associação 1 para Muitos.

Deve-se notar que a mesma associação existente entre as classes Socio e Dependente existe também entre as tabelas, mantendo-se a mesma multiplicidade e, devido a esta ser muitos na extremidade da tabela Dependente, força a criação da chave estrangeira nesta tabela. Observe que o relacionamento desta associação é feito entre a chave estrangeira da tabela Dependente e a chave primária da tabela Socio, sendo que, para que se possa descobrir o sócio ao qual o dependente está associado é necessário que o valor da chave estrangeira da linha do dependente em questão seja igual ao da chave primária de uma linha da tabela Socio.

Associação de Muitos para Muitos

Quando ocorre uma associação de muitos para muitos, deve-se criar uma tabela intermediária contendo duas chaves estrangeiras, cada uma relacionada à chave primária de uma das classes com a qual se relaciona. Essas chaves estrangeiras podem constituir a própria chave primária da tabela intermediária ou pode-se criar uma chave primária própria. A figura 4.45 apresenta um exemplo que ilustra uma situação como essa.

Esse exemplo foi adaptado a partir do modelo conceitual do sistema de controle bancário já apresentado anteriormente neste capítulo, onde uma pessoa pode possuir muitas contas e uma conta pode pertencer a mais de uma pessoa.

Uma associação muitos para muitos é perfeitamente aceitável dentro da orientação a objetos, desde que não haja atributos particulares relativos a um objeto específico associado com outro objeto específico, ou seja, atributos ligados à associação, caso em que é necessário incluir uma classe intermediária ou associativa, como já foi exposto. Em um modelo relacional, porém, uma associação

muitos para muitos exige a criação de uma tabela intermediária posicionada entre as tabelas envolvidas na associação.

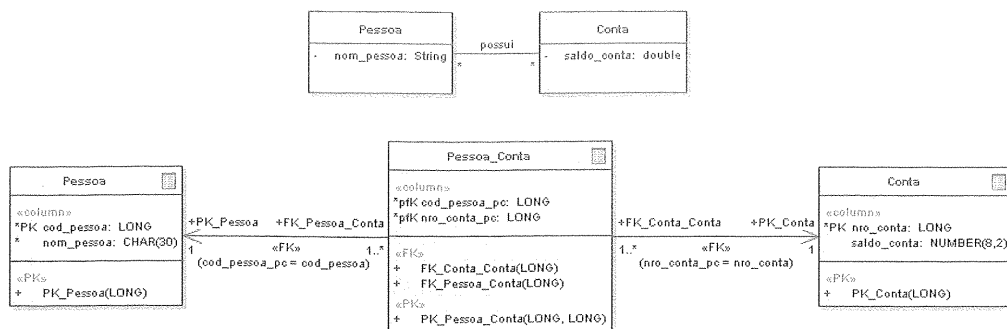


Figura 4.45 – Mapeamento de Classes com Associação Muitos para Muitos.

Ao observarmos a figura, percebemos que sua parte superior apresenta as classes **Pessoa** e **Conta**, ligadas por uma associação binária com multiplicidade muitos nas duas extremidades. Já na parte inferior da figura são apresentadas as tabelas mapeadas a partir dessas classes. O leitor perceberá que existem três tabelas para apenas duas classes, o que é necessário porque não é possível inserir uma coluna para referenciar cada conta possuída por uma pessoa, tampouco é possível acrescentar uma coluna para referenciar cada pessoa que possui uma determinada conta. Assim é necessário criar uma tabela intermediária para ligar cada pessoa a cada conta por ela possuída.

O leitor perceberá que existe uma tabela equivalente à classe **Pessoa** e outra à classe **Conta**, acrescentando-se somente uma chave primária para cada uma delas. Porém, existe uma tabela intermediária entre essas duas tabelas chamada **Pessoa_Conta**, contendo dois atributos denominados `cod_pessoa_pc` e `nro_conta_pc`. Esses atributos fazem as vezes tanto de chave primária da tabela como de chaves estrangeiras para pesquisar respectivamente uma pessoa e uma conta em particular, como demonstra o texto `pkf` na frente do nome das duas colunas. Observe que na terceira divisão da tabela está discriminado que esta contém duas chaves estrangeiras e uma chave primária composta por dois valores do tipo `LONG`.

O leitor notará que a tabela **Pessoa_Conta** está relacionada tanto à tabela **Pessoa** como à tabela **Conta** e que o relacionamento entre as tabelas **Pessoa_Conta** e **Pessoa** é feito por meio da comparação entre a chave primária `cod_pessoa` da tabela **Pessoa** e a chave estrangeira `cod_pessoa_pc` da tabela **Pessoa_Conta**, sendo necessário que os valores de ambas sejam iguais para se descobrir a qual pessoa pertence uma determinada conta. O mesmo ocorre entre as tabelas **Pessoa_Conta** e **Conta**, onde o relacionamento é feito entre a chave primária `nro_conta` da tabela **Conta** e a chave estrangeira `nro_conta_pc` da tabela **Pessoa_Conta**.

Quando já existe uma classe associativa ou intermediária, normalmente gera-se uma tabela para essa classe, contendo seus atributos. Em seguida, cria-se uma chave estrangeira para relacionar-se com cada chave primária das tabelas com a qual essa tabela se relaciona.

Associações Ternárias

No caso de associações ternárias utiliza-se um procedimento semelhante à situação de associações muitos para muitos, criando-se uma tabela para representar a associação ternária e uma chave estrangeira para esta se relacionar com cada tabela associada. A figura 4.46 apresenta um exemplo de mapeamento de associação ternária.

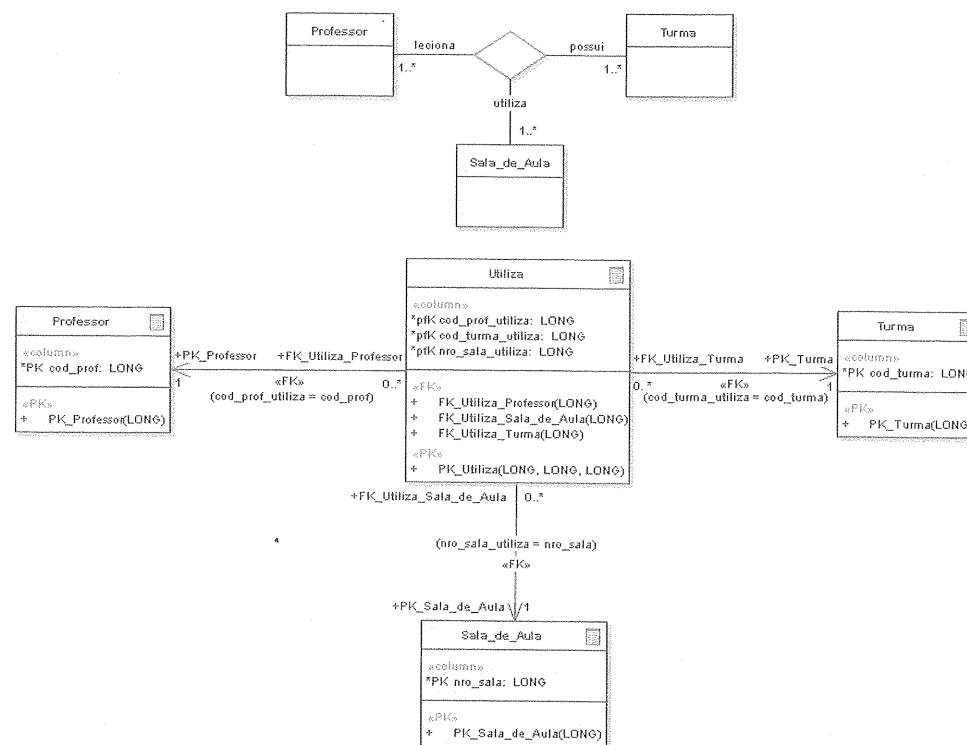


Figura 4.46 – Mapeamento de Associação Ternária.

Aqui tomamos o exemplo de associação ternária já apresentado no início deste capítulo e mapeamos essa associação. O leitor pode perceber que cada classe foi mapeada em uma tabela e que foi criada uma tabela intermediária mapeando a associação ternária em si, criando-se uma chave estrangeira para ligá-la a cada classe. Observe que a chave primária é composta pelas três colunas da tabela.

Herança

Quando se utiliza associações do tipo generalização/especialização, o processo de representação de tabelas relacionais é um pouco mais complexo. Existem basicamente três procedimentos possíveis.

Na primeira estratégia, toda a hierarquia de classes é representada por uma única tabela no banco de dados, que deve incluir uma coluna para identificar o tipo do objeto representado por cada linha. As desvantagens desta estratégia consistem na ausência de normalização dos dados e na possibilidade de existir muitos campos nulos em diversas linhas da tabela. A figura 4.47 apresenta um exemplo dessa estratégia.

Aqui identificamos uma situação em que existe uma classe abstrata denominada *Submissao*, que representa um conjunto de trabalhos submetidos a um evento científico. Como existem três tipos de submissões possíveis, referentes a artigos, propostas de minicursos ou de palestras, definiu-se a classe geral *Submissao* contendo os atributos comuns a todos os tipos de submissão e derivou-se três classes especializadas a partir dela, contendo cada uma seus atributos particulares.

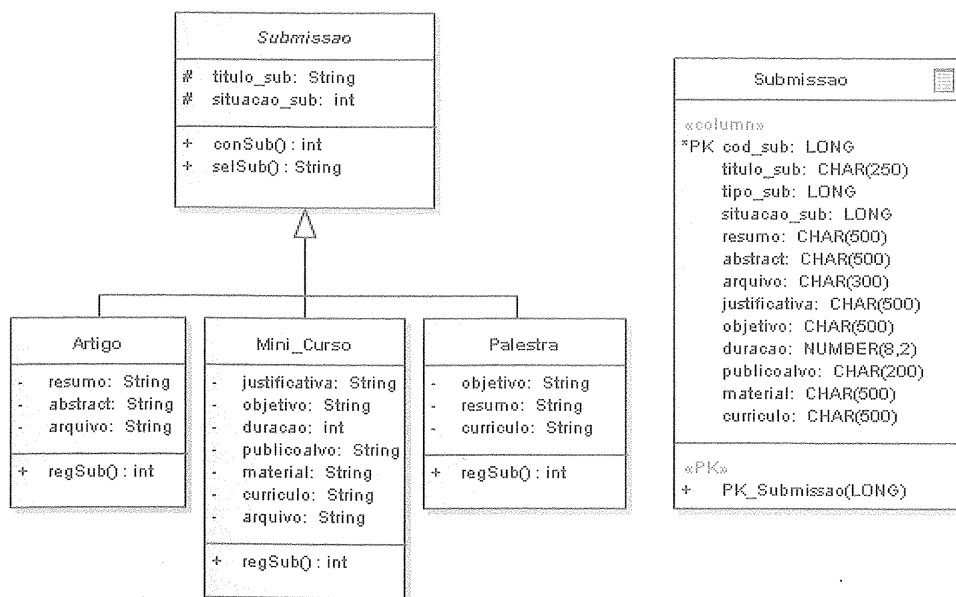


Figura 4.47 – Mapeamento de Herança – Primeira Estratégia.

O leitor notará que ao lado dessa hierarquia de classes existe uma tabela denominada *Submissao*, que foi mapeada a partir da referida hierarquia. Como aplicou-se nesse caso a primeira estratégia, existe uma tabela única para representar todas as classes da hierarquia, contendo colunas equivalentes a todos os

atributos identificados nas classes da referida hierarquia, além de uma coluna extra para identificar o tipo de submissão referente a cada linha da tabela, bem como uma coluna para servir de chave primária. Obviamente muitas colunas deverão ser deixadas em branco em cada registro da tabela.

Na segunda estratégia, cria-se uma tabela para cada classe concreta existente. Isso leva à redundância de dados, já que quaisquer atributos definidos em uma classe abstrata na hierarquia devem ser criados em todas as tabelas que representam classes-filhas da mesma. A figura 4.48 apresenta um exemplo dessa estratégia.

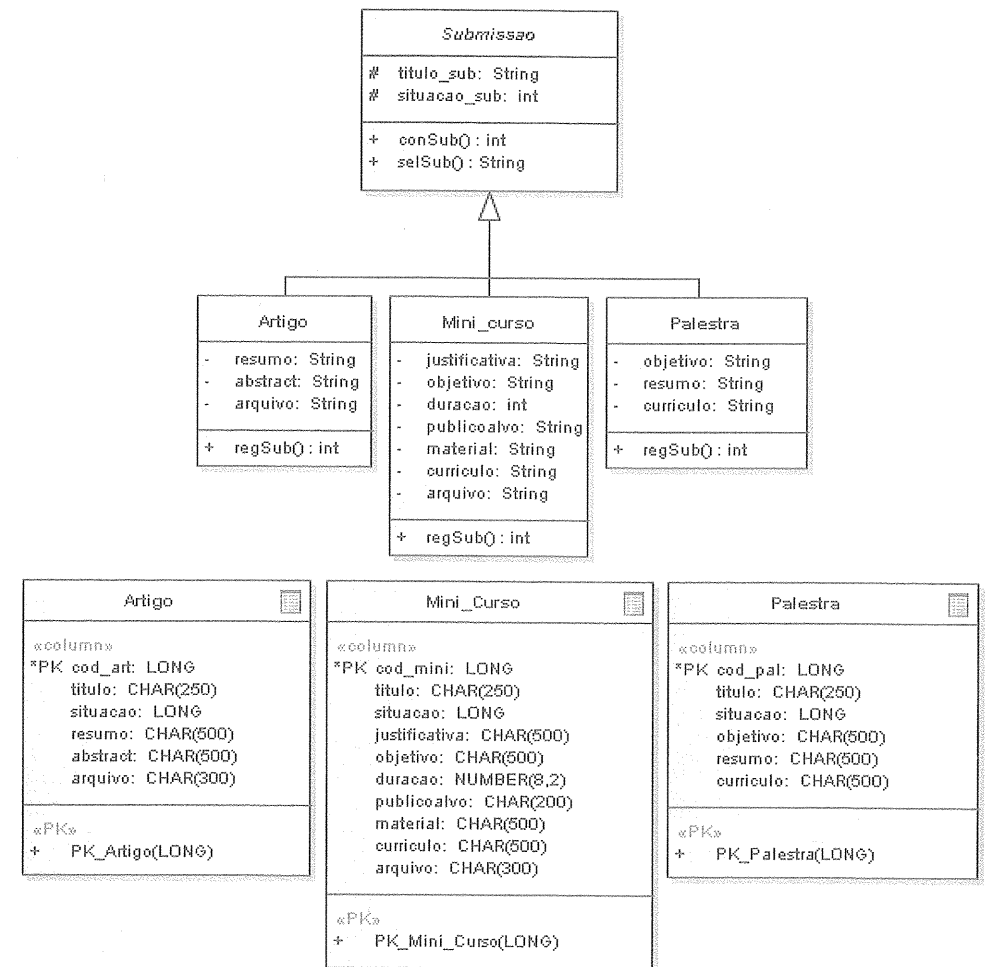


Figura 4.48 – Mapeamento de Herança – Segunda Estratégia.

Aqui tomamos o mesmo exemplo da figura anterior e aplicamos a segunda estratégia de mapeamento. Como o leitor poderá observar, criaram-se três tabelas, uma para armazenar os artigos submetidos, outra para armazenar os minicursos

e uma terceira para armazenar as propostas de palestras submetidas. Note que todas as tabelas têm as colunas *titulo* e *situacao*, que representam os atributos da superclasse *Submissao*.

Na terceira estratégia, cria-se uma tabela para cada classe da hierarquia, relacionando-as por meio de chaves estrangeiras. Essa estratégia tenta manter a normalização de dados, de forma que a estrutura final das tabelas fica bastante parecida com a hierarquia das classes. A figura 4.49 apresenta um exemplo dessa estratégia.

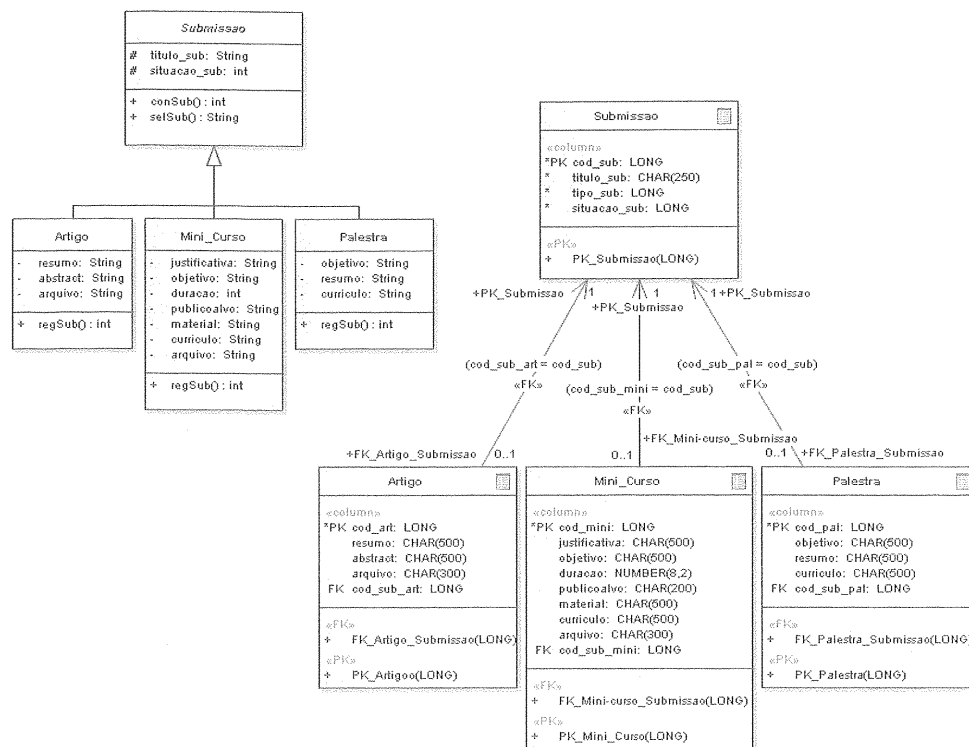


Figura 4.49 – Mapeamento de Herança – Terceira Estratégia.

Neste exemplo criou-se uma tabela *Submissao* que armazena os dados comuns a todas as submissões e uma tabela para cada tipo de submissão com suas colunas particulares. Observe que existe um relacionamento de 1 para 1 entre essas tabelas e a tabela *Submissao*, onde as tabelas *Artigo*, *Mini_Curso* e *Palestra* contêm uma chave estrangeira para relacionar-se com a tabela *Submissao*. Essas associações significam que uma submissão pode representar nenhum (0) ou um artigo específico, nenhum ou um determinado minicurso ou nenhuma ou uma palestra específica. Nessa estratégia não há redundância de dados, e o mapeamento é o mais semelhante à hierarquia de classes.

A seguir mapearemos o modelo conceitual do sistema de controle bancário apresentado anteriormente, conforme demonstra a figura 4.50.

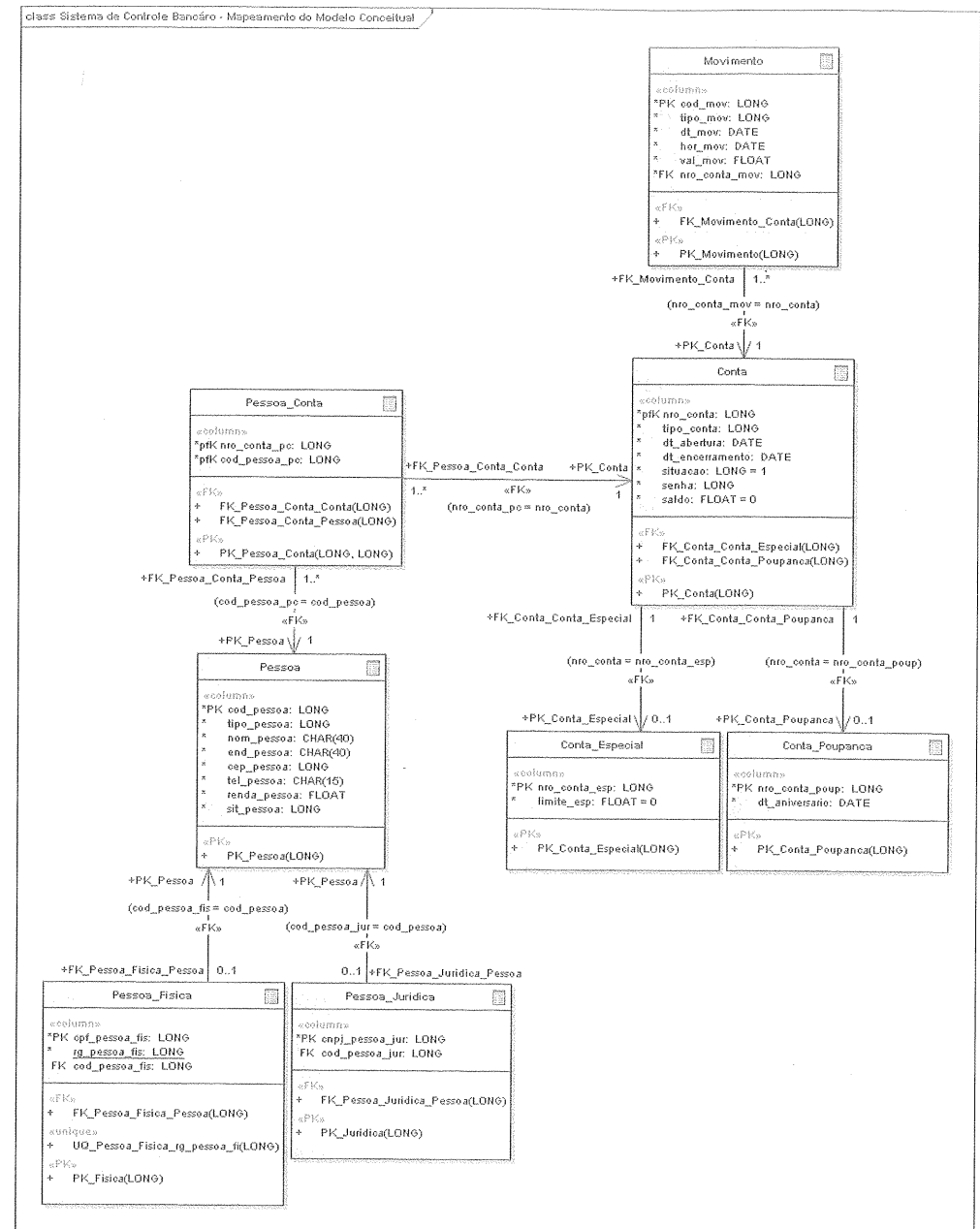


Figura 4.50 – Mapeamento do Modelo Conceitual do Sistema de Controle Bancário.

Para realizar esse mapeamento, em relação à hierarquia de classes optamos pela terceira estratégia. Dessa forma mapeamos uma tabela *Pessoa* que

corresponde à classe Pessoa do modelo conceitual e duas tabelas Pessoa_Fisica e Pessoa_Juridica que correspondem às classes derivadas a partir da classe Pessoa. Observe que a tabela Pessoa contém colunas correspondentes a todos os atributos da classe Pessoa, adicionando ainda uma coluna para representar a chave primária da tabela, chamada `cod_pessoa`, e uma coluna para identificar o tipo de pessoa, necessária para determinar se uma linha se refere a uma pessoa física ou jurídica.

Da mesma forma, as tabelas Pessoa_Fisica e Pessoa_Juridica contêm colunas correspondentes aos atributos das classes que elas representam, porém não foi necessário criar uma coluna para identificar a chave primária, uma vez que essa função pode ser desempenhada pela coluna que representa o CPF, na tabela Pessoa_Fisica, e pela que representa o CNPJ, na tabela Pessoa_Juridica. No entanto, foi necessário acrescentar uma coluna para identificar o código da pessoa que está associada à pessoa física ou jurídica, servindo esta coluna como chave estrangeira.

Em seguida criamos uma associação entre as tabelas Pessoa_Fisica e Pessoa, o mesmo ocorrendo entre as tabelas Pessoa_Juridica e Pessoa. Essas associações relacionam uma pessoa física a uma pessoa por meio da comparação do valor da chave estrangeira `cod_pessoa_fis` da tabela Pessoa_Fisica com a chave primária `cod_pessoa` da tabela Pessoa, ocorrendo coisa semelhante entre as tabelas Pessoa_Juridica e Pessoa, em que comparamos a chave estrangeira `cod_pessoa_jur`, da tabela Pessoa_Juridica, com a chave primária `cod_pessoa`, da tabela Pessoa. Observe que a multiplicidade dessas associações é 0..1 para 1, significando que uma pessoa física ou jurídica está relacionada a uma e somente uma pessoa, e uma pessoa pode estar relacionada a nenhuma ou uma pessoa física, ou a nenhuma ou uma pessoa jurídica. Uma pessoa pode estar associada a nenhuma pessoa física porque pode estar associada a uma pessoa jurídica e vice-versa, sendo que as pessoas física e jurídica são mutuamente exclusivas.

O mapeamento das classes Conta_Comum, Conta_Especial e Conta_Poupanca foi um pouco diferente devido ao fato de a classe Conta_Comum ser uma classe concreta e não abstrata. Dessa forma, mapeamos cada classe em uma tabela correspondente, nomeando a tabela relativa à classe Conta_Comum simplesmente Conta e inserindo-lhe uma coluna chamada `tipo_conta` para determinar se a linha se refere a uma conta comum, especial ou poupança. Não foi necessário inserir uma coluna para representar a chave primária, uma vez que a coluna `nro_conta` ocupa essa função perfeitamente.

Para manter a coerência com o modelo conceitual, relacionamos as tabelas Conta_Especial e Conta_Poupanca com a tabela Conta. O leitor poderá perceber que foi criada uma chave primária correspondente ao número da conta em ambas as tabelas.

Nesses relacionamentos, porém, diferentemente do que ocorre entre a tabela Pessoa e as tabelas Pessoa_Fisica e Pessoa_Juridica, as chaves estrangeiras estão posicionadas na tabela Conta, o que foi necessário porque a pesquisa sempre se dará na tabela Conta e, se for percebido que o tipo de conta não se refere a uma conta comum, será pesquisado, por meio do relacionamento, o limite da conta, se esta for especial, ou a data de aniversário, se for poupança. Observe que a coluna `nro_conta`, além de servir como chave primária na tabela Conta, serve também como chave estrangeira para pesquisar uma conta especial ou poupança, se isto for necessário.

Observe que a multiplicidade dessas associações é 1 para 0..1, o que significa que uma conta pode estar associada a nenhuma ou somente uma conta especial e que uma conta especial tem de estar associada a somente uma conta, o mesmo ocorrendo com a associação entre as tabelas Conta e Conta_Poupanca, uma vez que uma conta pode referir-se a uma conta especial ou a uma conta poupança ou a nenhuma das duas, quando se constitui em uma conta comum.

Assim, se uma linha da tabela Conta corresponder ao tipo conta comum, não haverá ligação nem com a tabela Conta_Especial nem com a tabela Conta_Poupanca. Porém, se o tipo for conta especial ou conta poupança, haverá uma ligação com uma linha da tabela correspondente.

Em seguida foi feito o mapeamento relativo à associação muitos para muitos entre as classes Pessoa e Conta_Comum no modelo conceitual, o que forçou a criação de uma tabela intermediária chamada Pessoa_Conta, que tem como chave primária duas colunas que armazenam o número da conta e o código da pessoa. Essas colunas também desempenham a função de chaves estrangeiras, permitindo assim relacionar cada linha dessa tabela com a pessoa e conta a qual ela corresponde.

Finalmente, foi mapeada a classe Movimento em uma tabela de mesmo nome, contendo colunas correspondentes a todos os seus atributos, adicionando ainda uma coluna para servir de chave primária e outra para servir de chave estrangeira, de maneira que possa ser possível determinar a qual conta cada movimento corresponde.

4.13 Exercícios Propostos

Esta seção dará continuidade à modelagem dos sistemas iniciados no capítulo anterior, desta vez enfocando a visão estrutural e estática do diagrama de classes.

4.13.1 Sistema de Controle de Cinema

Desenvolva o diagrama de classes para um sistema de controle de cinema, com base nos seguintes requisitos:

- Um cinema pode ter muitas salas, sendo necessário, portanto, registrar informações a respeito de cada sala, como sua capacidade, ou seja, o número de assentos disponíveis.
- O cinema apresenta muitos filmes. Um filme tem informações como título e duração. Assim, sempre que um filme for ser apresentado, deve-se registrá-lo também.
- Um filme tem um único gênero, mas um gênero pode se referir a muitos filmes.
- Um filme pode ter muitos atores atuando nele, e um ator pode atuar em muitos filmes. Em cada filme, um ator interpretará um ou mais papéis diferentes. Por uma questão de propaganda, é útil anunciar os principais atores do filme e que papéis eles interpretam.
- Um mesmo filme pode ser apresentado em diferentes salas e horários. Cada apresentação em uma determinada sala e horário é chamada Sessão. Um filme sendo apresentado em uma sessão tem um conjunto máximo de ingressos, determinado pela capacidade da sala.
- Os clientes do cinema podem comprar ou não ingressos para assistir a uma sessão. O funcionário deve intermediar a compra do ingresso. Um ingresso deve conter informações como o tipo de ingresso (meio ingresso ou ingresso inteiro). Além disso, um cliente só pode comprar ingressos para sessões ainda não encerradas.

4.13.2 Sistema de Controle de Clube Social

Desenvolva o diagrama de classes para um sistema de controle de clube social de acordo com os seguintes requisitos:

- O clube tem muitos sócios e precisa manter informações referentes a eles, como o número de seu cartão de sócio, nome, endereço, telefone e e-mail.
- Um sócio pode ter nenhum ou muitos dependentes, mas um dependente está associado a somente um sócio. O clube precisa manter informações sobre os dependentes de cada sócio, como o número de seu cartão, nome, parentesco e e-mail.
- Um sócio deve pertencer a uma única categoria. No entanto, pode haver muitos sócios pertencentes a uma determinada categoria.
- Um sócio deve pagar mensalidades para poder frequentar o clube. Assim, enquanto permanecer sócio do clube, um sócio pode pagar muitas mensalidades, mas uma mensalidade pertence a somente um sócio. Eventualmente um sócio pode não estar adimplente. Nesse caso, serão cobrados

juros sobre o valor da mensalidade relativos ao atraso do pagamento. É também possível que um sócio nunca tenha pago suas mensalidades. As informações pertinentes a cada mensalidade são a data de pagamento, o valor, a data em que foi efetivamente paga, os possíveis juros aplicados, o valor efetivamente pago e se está quitada ou não.

4.13.3 Sistema de Locação de Veículos

Desenvolva o diagrama de classes para um sistema de locação de veículos, levando em consideração os seguintes requisitos:

- A empresa tem muitos automóveis. Cada automóvel tem atributos como número da placa, cor, ano, tipo de combustível, número de portas, quilometragem, renavam, chassi, valor de locação etc.
- Cada carro tem um modelo e uma marca, mas um modelo pode relacionar-se a muitos carros e uma marca pode referir-se a muitos modelos, embora cada modelo só tenha uma marca específica.
- Um carro pode ser alugado por muitos clientes, em momentos diferentes, e um cliente pode alugar muitos carros. É preciso saber quais carros estão locados ou não. Sempre que um carro for locado é preciso armazenar a data e a hora de sua locação e, quando for devolvido, a data e hora de devolução.

4.13.4 Sistema para Controle de Leilão Via Internet

Desenvolva o diagrama de classes para um sistema de leilão via internet, de acordo com os seguintes requisitos:

- Cada leilão deve conter informações como data de início, hora de início, data de encerramento e hora de encerramento.
- Em cada leilão existem diversos itens a serem leiloados. Cada item está associado a um único leilão. Se não for leiloadado naquele momento, deverá ser cadastrado como item de outro leilão novamente. Cada item tem um lance mínimo.
- Um item pode receber muitos lances, mas pode não receber nenhum. Nesse último caso não será arrematado.
- Existem diversos participantes em cada leilão interessados em adquirir os itens ofertados. Os participantes devem se registrar via internet, antes de o leilão iniciar.
- Um participante pode realizar quantos lances quiser, mas não é obrigado a realizar lance algum.

4.13.5 Sistema de Controle de Hotelaria

Desenvolva o diagrama de classes para um sistema de controle de hotelaria de acordo com os seguintes fatos:

- O hotel aluga quartos de diversas categorias (simples, duplo, casal, luxo etc.). O valor dos quartos varia de acordo com a categoria.
- Cada hóspede precisa ser identificado no momento em que ocupa um quarto, mesmo que este seja pago por outro hóspede. Caso seu cadastro ainda não exista ou seus dados tenham mudado, é necessário cadastrá-lo.
- Um hóspede pode alugar muitos quartos, em um mesmo momento ou em momentos diferentes, e um quarto pode ser alugado por muitos hóspedes, em momentos diferentes, naturalmente.
- Dependendo da categoria do quarto, ele terá uma determinada quantidade de itens, tanto no quarto propriamente dito como no frigobar.
- Um hóspede pode consumir itens do frigobar. Cada item tem valores e quantidades diferentes.
- Um hóspede pode solicitar serviços do hotel.
- Cada quarto gera diárias sempre ao meio-dia. Uma diária deve ser paga exclusivamente por um determinado hóspede, mas um hóspede pode pagar muitas diárias.
- É necessário saber qual funcionário foi responsável pela locação e/ou encerramento de cada locação de um quarto.

4.14 Solução dos Exercícios

Nesta seção apresentaremos as soluções dos problemas propostos. Aqui cumpre chamar a atenção para a declaração de alguns métodos das soluções apresentadas. A rigor é necessário haver um get e um set para cada método de uma classe, mas é inviável representar esses métodos nesses diagramas devido ao grande espaço que ocupariam. Por esse motivo muitas vezes declaramos métodos “genéricos” para retornar várias informações de uma classe e definimos seu retorno como uma String.

4.14.1 Sistema de Controle de Cinema

A figura 4.51 apresenta a solução desse exercício. Pode-se perceber que o diagrama apresentado já se refere ao modelo de domínio.

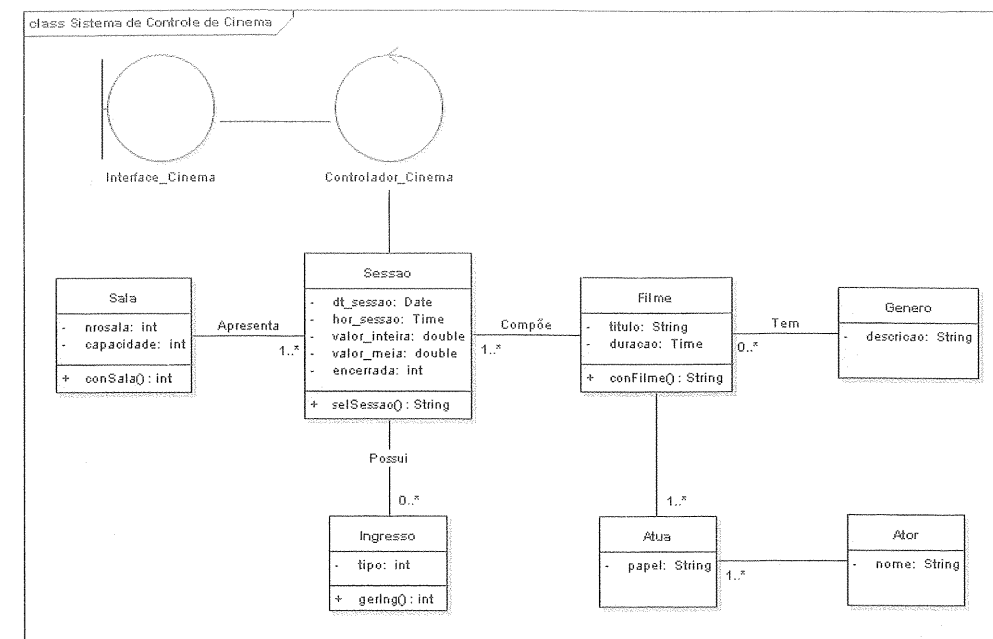


Figura 4.51 – Diagrama de Classes para o Sistema de Controle de Cinema – Solução do Exercício.

Inserimos uma classe de fronteira para representar a interface do sistema e uma classe controladora para interpretar os eventos da interface e solicitar o disparo de métodos nas classes de entidade. A seguir explicaremos as classes de entidade que compõem o diagrama. Nesse diagrama identificamos somente os métodos considerados mais importantes.

1. Genero

Essa classe armazena os gêneros de filmes apresentados pelo cinema. Seu único atributo é a descrição do gênero do tipo String.

2. Filme

Essa classe contém o título e a duração de cada filme apresentado no cinema. Seu único método serve para consultar os filmes cadastrados. Observe que um filme está associado a um único gênero, mas um gênero pode estar associado a muitos filmes.

3. Ator

Essa classe representa os atores que interpretam papéis nos filmes exibidos no cinema. Seu único atributo é o nome do ator.

4. Atua

Esta é uma classe intermediária entre as classes Ator e Filme e representa os papéis que um ator interpreta em cada filme em que participa. Seu único atributo consiste no papel que um ator interpreta. Observe que um ator pode atuar em no mínimo um e no máximo muitos filmes e que um filme pode ter muitos atores atuando nele, sendo que um filme deve ter pelo menos um ator.

5. Sala

Essa classe armazena as informações sobre as salas pertencentes ao cinema. Seus atributos são número da sala e capacidade da sala, ambos do tipo inteiro. Seu único método permite consultar uma determinada sala e retornar sua capacidade.

6. Sessao

Essa classe representa as sessões de filmes apresentadas pelo cinema. Seus atributos são data da sessão, hora da sessão, valor do ingresso inteiro, valor do meio ingresso e um atributo para determinar se a sessão já foi encerrada ou não. Observe que uma sessão é apresentada em uma única sala, mas que em uma sala podem ter sido apresentadas muitas sessões. Note também que em uma sessão é exibido somente um filme, mas um filme pode ser exibido em muitas sessões. O único método definido nessa classe permite selecionar as sessões ainda em aberto, retornando uma String com os dados da sessão em questão.

7. Ingresso

Finalmente, essa classe representa os ingressos vendidos em cada sessão de cinema. Esta classe armazena o tipo de ingresso, se é um ingresso inteiro ou meio ingresso, tendo ainda um método para gerar um novo ingresso, que retorna verdadeiro (1), se foi possível gerar, ou falso (0), em caso contrário. Uma sessão pode gerar muitos ingressos, contudo, pode não gerar ingresso algum.

4.14.2 Sistema de Controle de Clube Social

A solução desse exercício é apresentada na figura 4.52. Como no exercício anterior, esse diagrama já se refere ao modelo de domínio e, da mesma forma que no exercício anterior, somente foram identificados os métodos considerados pertinentes.

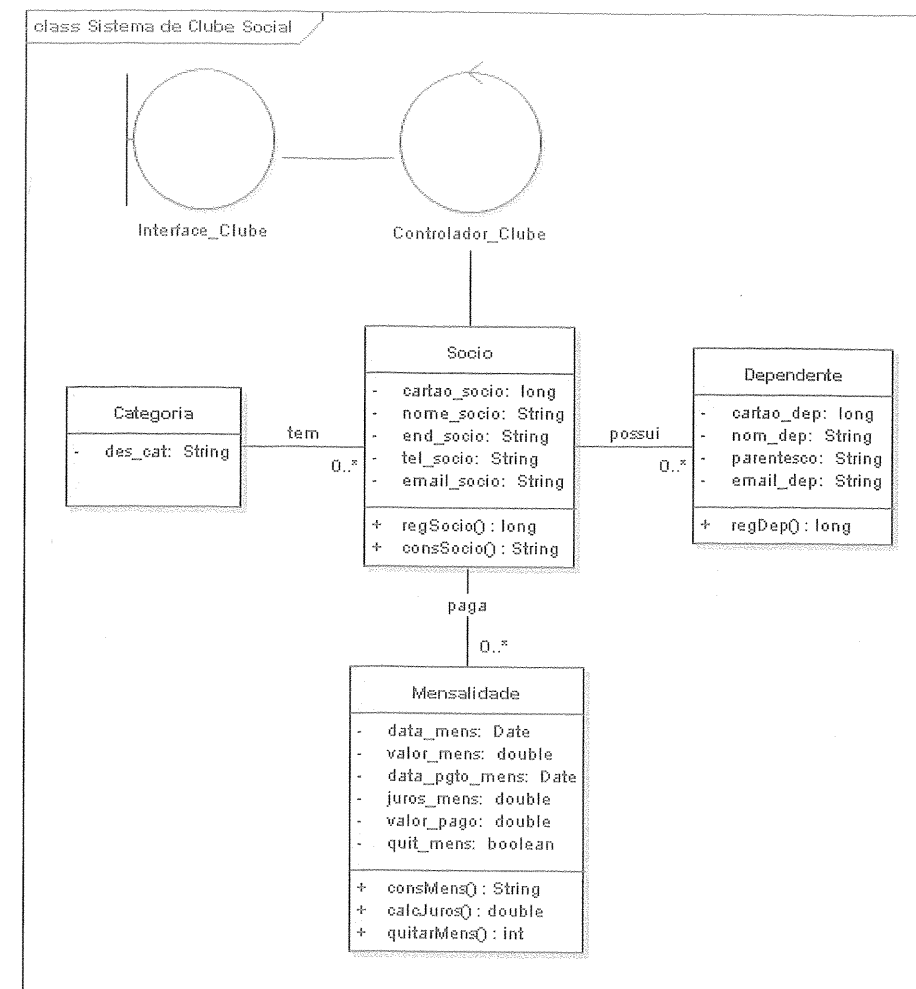


Figura 4.52 – Diagrama de Classes para o Sistema de Controle de Clube Social – Solução do Exercício.

Também igualmente como no exercício anterior inserimos uma classe de fronteira e uma controladora. A seguir explicaremos as classes de entidade.

1. Categoria

Essa classe representa as possíveis categorias de sócios estabelecidas pelo clube. Seu único atributo é a descrição da categoria.

2. Socio

Essa classe armazena as informações referentes aos sócios do clube. Os atributos dessa classe são autoexplicativos. A classe tem dois métodos, um para registrar um sócio e outro para consultar um sócio específico.

O método `regSocio` retorna um `long` que representa o número do cartão do sócio, e o método `consSocio` retorna uma `String` contendo os dados de um determinado cliente. Observe que um sócio pertence a uma categoria, mas uma categoria pode estar associada a muitos sócios.

3. Dependente

Essa classe armazena as informações referentes aos possíveis dependentes de um sócio. Os atributos da classe são autoexplicativos. O único método contido pela classe permite gerar uma nova instância da mesma. Pode-se perceber que um dependente está relacionado a um único sócio, mas um sócio pode não ter nenhum dependente ou pode ter vários.

4. Mensalidade

A classe em questão representa as mensalidades que devem ser pagas por cada sócio. Seus atributos são data da mensalidade e data em que a mensalidade foi efetivamente paga, do tipo `Date`, valor da mensalidade, juros da mensalidade e o valor efetivamente pago do tipo `double`, e um atributo denominado `quit_mens` do tipo `boolean`, que determina se a mensalidade foi quitada ou não. Já os métodos da classe são:

Método	Descrição
<code>consMens</code>	É disparado para consultar cada mensalidade ainda não paga de um determinado sócio. Retorna uma <code>String</code> com os dados da mensalidade.
<code>calcJuros</code>	Calcula os juros de uma mensalidade, no caso de esta estar atrasada. Retorna um <code>double</code> contendo o valor atual, após a aplicação de juros, da mensalidade.
<code>quitarMens</code>	Permite a quitação de uma mensalidade, retornando verdadeiro, se a operação foi concluída com sucesso, ou falso, se ocorreu algum problema quando se tentou quitar a mensalidade.

4.14.3 Sistema de Locação de Veículos

A solução desse exercício é apresentada na figura 4.53. Da mesma forma que no exercício anterior, esse diagrama já se refere ao modelo de domínio e igualmente só foram identificados os métodos considerados pertinentes.

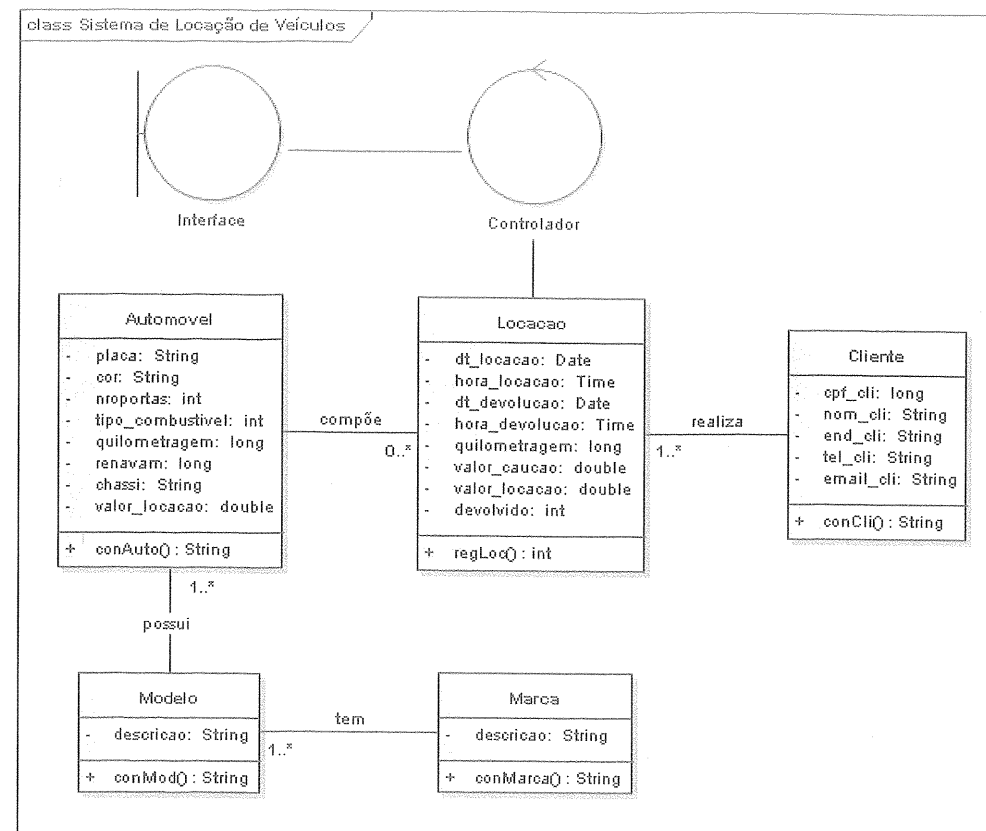


Figura 4.53 – Diagrama de Classes para o Sistema de Locação de Veículos – Solução do Exercício.

Do mesmo modo que no exercício anterior, foi criada uma classe de fronteira e uma controladora. A seguir explicaremos as classes de entidade desse diagrama.

1. Marca

Essa classe representa as marcas dos carros disponíveis na locadora. Tem como atributo a descrição da marca, do tipo `String`, além do método `consMarca`, usado para consultar uma marca específica. Como pode-se perceber, o método retorna uma `String`.

2. Modelo

Já essa classe armazena os modelos das marcas disponíveis na locadora. À semelhança da classe `Marca`, essa classe tem como atributo a descrição do modelo e um método que permite consultar um determinado modelo. Observe que um modelo tem somente uma marca, mas uma marca pode ter muitos modelos.

3. Cliente

Essa classe, por sua vez, armazena as informações relativas aos clientes que locam veículos. Seus atributos são autoexplicativos. A classe tem ainda um método para consultar um determinado cliente e retornar seus dados.

4. Automovel

Essa classe contém as informações dos automóveis locados pela empresa, indo desde a placa do veículo até o valor de locação. Existe ainda um método que permite consultar um determinado automóvel, retornando os dados deste. Um automóvel está relacionado a um único modelo, mas um modelo pode estar associado a muitos automóveis.

5. Locacao

Essa classe apresenta as informações relativas às locações de automóveis já realizadas. A classe tem os atributos data de locação e data de devolução, do tipo Date; hora de locação e hora de devolução, do tipo Time; quilometragem, do tipo long; valor de caução e de locação, do tipo double; e um atributo chamado “devolvido”, do tipo int, cuja função é determinar se a locação foi devolvida ou não. A classe tem ainda um método para registrar uma locação, que retorna verdadeiro se for executado com sucesso e falso em caso contrário. Ao examinarmos as associações dessa classe, concluímos que um cliente pode realizar muitas locações (no mínimo uma), mas que uma locação refere-se a um único cliente, e que uma locação refere-se a um veículo em particular, mas que um mesmo veículo pode estar relacionado a muitas locações ou não ter sido locado nunca.

4.14.4 Sistema para Controle de Leilão Via Internet

A solução desse exercício é apresentada na figura 4.54. Novamente, como no exercício anterior, este diagrama já se refere ao modelo de domínio e novamente só foram identificados os métodos considerados pertinentes.

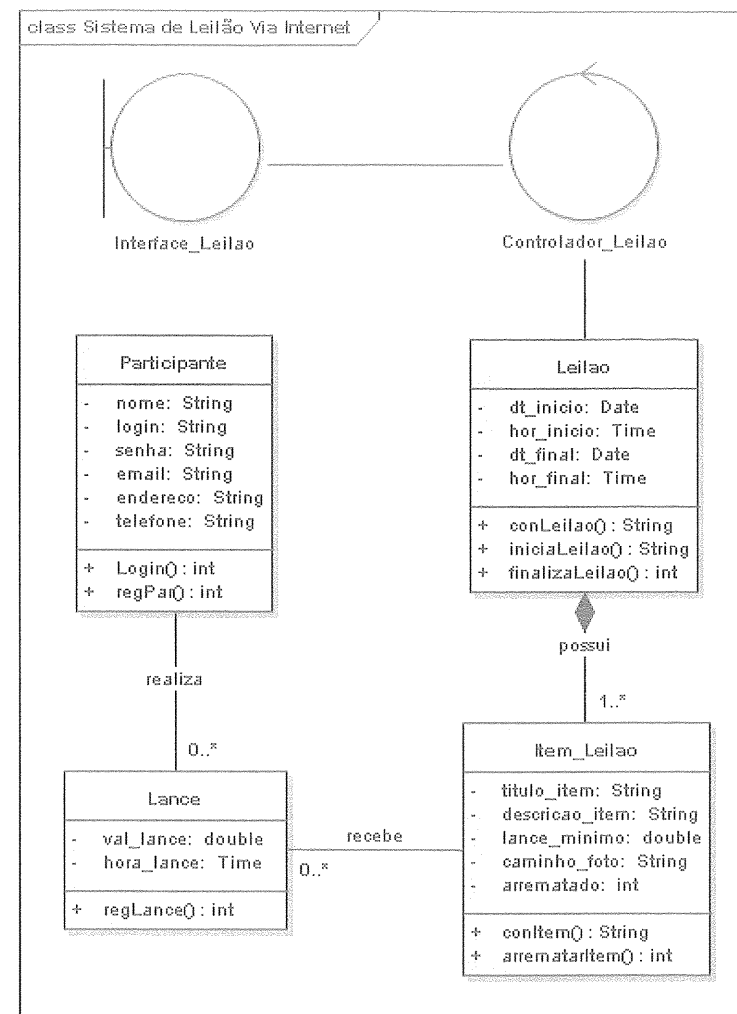


Figura 4.54 – Diagrama de Classes para o Sistema de Controle de Leilão Via Internet – Solução do Exercício.

Como de praxe, foram criadas uma classe de fronteira e uma classe controladora. A seguir explicaremos as classes de entidade contidas neste diagrama.

1. Participante

Essa classe refere-se às informações das pessoas que se cadastraram para participar do leilão e possivelmente oferecer lances para os itens leiloados. Os atributos da classe são autoexplicativos. A classe contém ainda os métodos para permitir a um participante se logar, denominado Login, e para permitir o autorregistro de um novo participante, chamado regPar. Ambos os métodos retornam verdadeiro, se o método atingiu seu objetivo, ou falso, em caso contrário.

2. Leilao

Aqui identificamos nesta classe os leilões programados ou realizados pela instituição. Seus atributos são a data de início e fim do leilão, do tipo Date, bem como a hora de início e fim, do tipo Time. Essa classe tem ainda os seguintes métodos:

Método	Descrição
conLeilao	É disparado para consultar os leilões registrados, retornando uma String com os dados referentes a cada leilão.
iniciaLeilao	Tem por função iniciar um leilão. Por depender do retorno de outro método de outra classe, retorna uma String.
finalizaLeilao	Serve para marcar um leilão específico como finalizado. Retorna verdadeiro se consegue realizar seu intento e falso em caso contrário.

3. Item_Leilao

Essa classe contém as informações dos itens que serão ofertados em cada leilão. Seus atributos são título, descrição e caminho da foto do item, do tipo String, lance mínimo, do tipo double, e um atributo identificado como “arrematado”, do tipo int, cujo objetivo é determinar se o item foi arrematado (1) ou não (0).

Observe que a classe Leilao tem uma associação de composição com a classe Item_Leilao, o que significa que a informação de um objeto Leilao deve ser complementada por um ou mais objetos Item_Leilao e que um item de leilão está associado exclusivamente a um leilão específico. Essa relação todo-parte é particularmente clara ao examinarmos os métodos IniciaLeilao e ConItem, principalmente se consultarmos o diagrama de sequência referente ao processo Realizar Leilão, no final do capítulo sobre o diagrama de sequência.

A classe Item_Leilao contém os seguintes métodos:

Método	Descrição
conItem	É disparado para consultar um item específico, estando associado ao método IniciaLeilao da classe Leilao. Retorna uma String contendo os dados do item.
arrematarItem	Tem o objetivo de definir um determinado item como arrematado, retornando verdadeiro se for bem-sucedido e falso em caso contrário.

4. Lance

Essa classe é responsável por armazenar os lances realizados sobre cada item ofertado. Seus atributos são valor do lance, do tipo double, e o

momento em que o lance foi ofertado, do tipo Time. Seu único método, regLance, permite registrar um novo lance. Observe que um lance refere-se a um único item, mas que um item pode receber nenhum ou muitos lances. Além disso, um lance é ofertado por um participante específico, mas um participante pode realizar nenhum ou muitos lances.

4.14.5 Sistema de Controle de Hotelaria

Na figura 4.55 podemos examinar a solução desse exercício. Como nas outras soluções, esse diagrama já se refere ao modelo de domínio, embora um tanto incompleto, uma vez que nesse diagrama só foram declarados os métodos considerados pertinentes ao processo de Quitar Diárias.

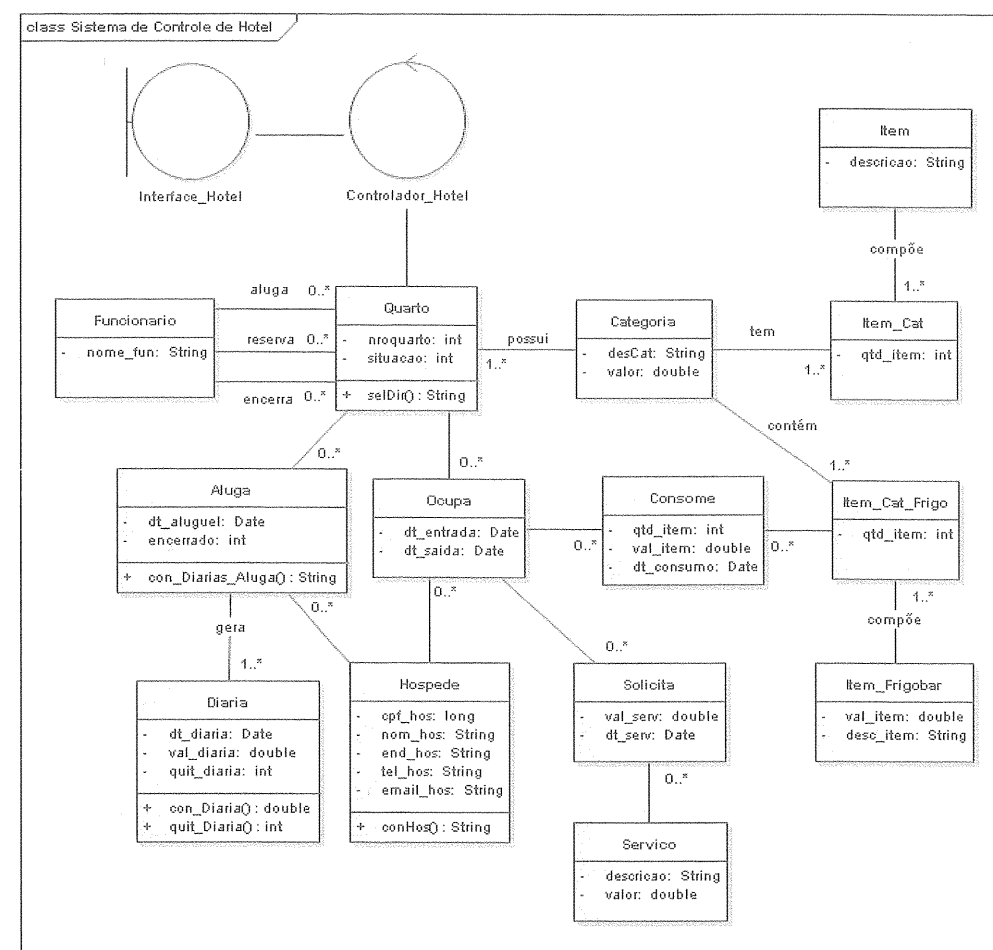


Figura 4.55 – Diagrama de Classes para o Sistema de Controle de Hotelaria – Solução do Exercício.

A exemplo dos outros exercícios, foram criadas uma classe de fronteira e uma classe controladora. A seguir explicaremos as classes de entidade contidas nesse diagrama.

1. Funcionario

Essa classe armazena as informações dos funcionários que trabalham ou trabalharam no hotel. Seu único atributo é o nome do funcionário.

2. Quarto

Aqui identificamos os quartos alugados pelo hotel. Os atributos dessa classe são o número do quarto e a sua situação, que determina se o quarto está livre (0), ocupado (1) ou reservado (2). Os dois atributos são do tipo `int`. O método `selecionar` desta classe é utilizado para selecionar as diárias de um determinado quarto e retorna uma `String` com essas informações. É importante notar que a classe `Funcionario` tem três associações com essa classe, uma vez que um funcionário pode reservar, alugar ou encerrar a estadia de um quarto para um hóspede e é necessário saber quais funcionários realizaram essas operações. Um quarto pode ser alugado, encerrado ou liberado por um funcionário específico, mas um mesmo funcionário pode realizar muitas dessas operações diversas vezes.

3. Categoria

A classe em questão contém as informações sobre as categorias de quartos disponíveis no hotel. O tipo de categoria influi no valor do quarto. Seus atributos são a descrição da categoria do tipo `String` e o valor a ser cobrado pelo aluguel de um quarto pertencente a uma categoria, do tipo `double`. Uma categoria pode pertencer a muitos quartos, mas um quarto pode ter somente uma categoria.

4. Item

Essa classe se refere aos itens que podem estar presentes em um quarto (isso varia de acordo com a categoria). Seu único atributo é a descrição do quarto do tipo `String`.

5. Item_Cat

Esta é uma classe intermediária entre as classes `Categoria` e `Item` e identifica os itens contidos nos quartos de uma determinada categoria. Seu único atributo é a quantidade de um determinado item, do tipo `int`, uma vez

que essa quantidade pode variar de categoria para categoria. Um objeto dessa classe está associado a um único objeto da classe `Item` e a um único objeto da classe `Categoria`, mas uma categoria pode ter muitos itens e um item pode estar associado a muitas categorias.

6. Item_Frigobar

Semelhante à classe `Item`, essa classe refere-se aos itens de frigobar que podem estar contidos em um quarto, o que também varia de acordo com a categoria. Seus atributos referem-se à descrição do item, do tipo `String`, e ao valor do item, do tipo `double`.

7. Item_Cat_Frigo

Já essa classe é semelhante à classe `Item_Cat`. Refere-se aos itens de frigobar contidos nos quartos de uma determinada categoria. Seu único atributo armazena a quantidade de um determinado item armazenado nos quartos de uma determinada categoria, e seu tipo é `int`.

8. Servico

Essa classe representa os serviços oferecidos pelo hotel. Seus atributos são a descrição do serviço, do tipo `String`, e o valor do serviço, do tipo `double`.

9. Hospede

O objetivo dessa classe é armazenar as informações de todos os hóspedes que já alugaram, reservaram e/ou ocuparam quartos no hotel (um quarto não necessariamente é ocupado por quem o aluga: uma instituição pode alugar um quarto para um palestrante, por exemplo). Seus atributos são autoexplicativos. A classe tem um método para consultar um determinado hóspede, que retorna uma `String` com seus dados.

10. Diaria

Essa classe armazena as diárias devidas por um hóspede. Seus atributos são a data a que a diária se refere, do tipo `Date`, o valor da diária, do tipo `double`, e o atributo `quit_diaria`, do tipo `int`, que determina se foi quitada (1) ou não (0). A classe tem um método para consultar uma diária, retornando um valor `double`, denominado `con_Diaria`, e um método para quitar uma diária, chamado `quit_Diaria`, que retorna um valor inteiro determinando se a operação pôde ser concluída ou não.

11. Aluga

Esta é uma classe intermediária entre as classes *Hospede* e *Quarto*, que armazena as informações referentes ao aluguel de um determinado quarto por um determinado hóspede, que não necessariamente irá ocupá-lo, podendo alugá-lo para outro hóspede. A classe contém o atributo referente à data do aluguel do tipo *Date* e o atributo “encerrado”, que determina se o aluguel se encontra encerrado (1) ou não (0), do tipo *int*. A classe contém um método denominado *con_Diarias_Aluga*, que retorna uma *String* e é utilizado para retornar os valores das diárias devidas pelo aluguel. Como podemos observar, um hóspede pode alugar muitos quartos e um quarto pode ser alugado por muitos hóspedes, mas um objeto da classe *Aluga* está associado exclusivamente a um único hóspede e a um único quarto.

12. Ocupa

Essa classe refere-se a um quarto efetivamente ocupado por um hóspede, armazenando os atributos data de entrada e data de saída, do tipo *Date*. Um hóspede pode ocupar muitos quartos, e um quarto pode ser ocupado por muitos hóspedes, mas um objeto da classe *Ocupa* está associado exclusivamente a um objeto da classe *Hospede* e a um objeto da classe *Quarto*.

13. Consome

Essa classe armazena as informações dos itens de frigobar consumidos por um hóspede na época em que ocupava um determinado quarto. Seus atributos são a quantidade consumida do item, do tipo *int*, seu valor na época, do tipo *double*, e a data em que o item foi consumido, do tipo *Date*. Um hóspede ocupando um quarto pode consumir nenhum ou muitos itens de frigobar, e um item de frigobar pode ser consumido por muitos hóspedes. Contudo, um objeto da classe *Consome* está associado única e exclusivamente a um objeto da classe *Ocupa* e a um objeto da classe *Item_Cat_Frigo*.

14. Solicita

Essa classe identifica os serviços porventura solicitados por um determinado hóspede ocupando um determinado quarto. Seus atributos são o valor do serviço na época em que foi solicitado, do tipo *double*, e a data em que este foi solicitado, do tipo *Date*. Um hóspede em um quarto pode solicitar nenhum ou muitos serviços, e um serviço pode ser solicitado por muitos hóspedes, mas um objeto da classe *Solicita* associa-se exclusivamente a um objeto da classe *Ocupa* e a um objeto da classe *Servico*.

CAPÍTULO 5

Diagrama de Objetos

O diagrama de objetos tem como objetivo fornecer uma “visão” dos valores armazenados pelos objetos das classes, definidas no diagrama de classes, em um determinado momento do sistema. Assim, embora o diagrama de classes seja estático, podem ser criados diagramas de objetos, onde as possíveis situações pelas quais os objetos das classes passarão possam ser simuladas.

5.1 Objeto

Um componente objeto é bastante semelhante a um componente classe, mas os objetos não apresentam métodos, somente atributos, e estes armazenam os valores contidos nos objetos em uma determinada situação. O nome dos objetos está contido, como nas classes, na primeira divisão do retângulo que representa os objetos e pode ser apresentado de três formas:

- O nome do objeto, com todas as letras minúsculas, seguido do símbolo de dois pontos (:) e o nome da classe à qual o objeto pertence, com as letras iniciais maiúsculas. Este é o formato mais completo.
- O nome do objeto omitido, mas mantendo o símbolo de dois pontos e o nome da classe.
- Somente o nome do objeto, sem dois pontos.

A figura 5.1 apresenta um exemplo de objeto chamado *pesfis1*, pertencente à classe *Pessoa_Fisica*.

Neste exemplo identificamos um objeto individual, pertencente à classe *Pessoa_Fisica* do sistema de controle bancário que temos estado a modelar. O leitor poderá perceber que os atributos do objeto em questão foram retirados tanto da classe *Pessoa* como da classe *Pessoa_Fisica*, uma vez que a última classe é derivada da primeira e, portanto, herda seus atributos. Observe que todos os atributos do objeto possuem valores, que foram definidos durante sua instânciação ou ao longo do tempo em que o objeto foi manipulado pelo sistema.

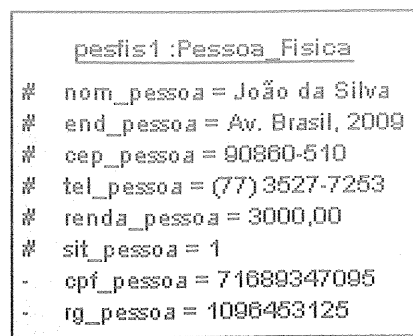


Figura 5.1 – Exemplo de Objeto.

5.2 Vínculos

Os objetos de um diagrama de objetos apresentam vínculos entre si (links). Esses vínculos nada mais são do que instâncias das associações entre as classes representadas no diagrama de classes, assim como os objetos são instâncias das próprias classes. Um vínculo tem exatamente o mesmo símbolo utilizado pelas associações do diagrama de classes, mas não apresenta multiplicidade porque esta especifica justamente o número de instâncias de uma determinada classe que podem estar envolvidas em uma associação. Assim, um vínculo em um diagrama de objetos liga apenas um único objeto em cada extremidade. A figura 5.2 apresenta um exemplo de vínculo entre objetos.

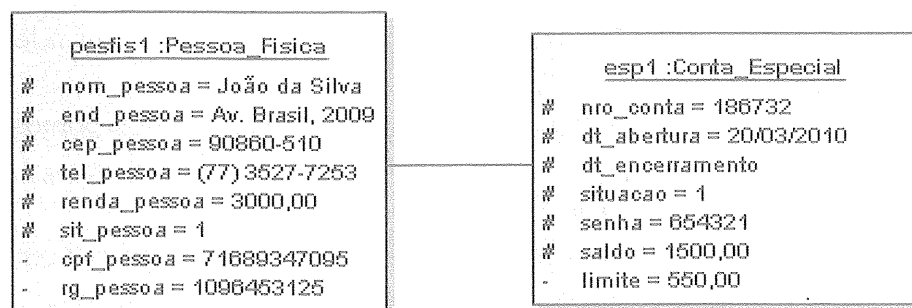


Figura 5.2 – Vínculo entre Objetos.

Nesse exemplo criamos uma visão por meio do diagrama de objetos referente ao vínculo entre um objeto da classe Pessoa_Fisica e um objeto da classe Conta_Especial a ele relacionado. Percebemos que o objeto pesfis1 está ligado ao objeto esp1. Observe que os atributos do objeto Conta_Especial foram retirados tanto da

classe Conta_Comum como da classe Conta_Especial, devido ao fato de essa última ser uma especialização da classe Conta_Comum. Observe que a situação da conta está ativa e, por esse motivo, a data de encerramento da conta não foi informada.

5.3 Dependência com Estereótipo <<instantiate>>

É possível representar os objetos instanciados a partir de classes por meio de uma associação de dependência junto com o estereótipo <<instantiate>>, como pode ser observado na figura 5.3.

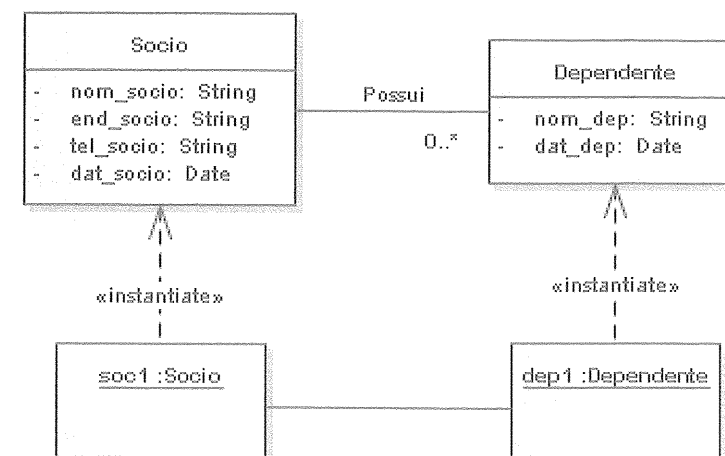


Figura 5.3 – Dependência com Estereótipo <<instantiate>>.

Nesse exemplo representamos a instanciação do objeto soc1 a partir da classe Socio e do objeto dep1 a partir da classe Dependente através da associação de dependência apoiada pelo estereótipo <<instantiate>>. Observe que a associação entre as classes é representada por um vínculo entre os objetos, mas sem multiplicidade, já que, como foi dito, um vínculo une um único objeto a outro.

5.4 Exemplo de Diagrama de Objetos

A figura 5.4 apresenta um exemplo de diagrama de objetos, criando uma visão das contas e movimentos de uma pessoa física.

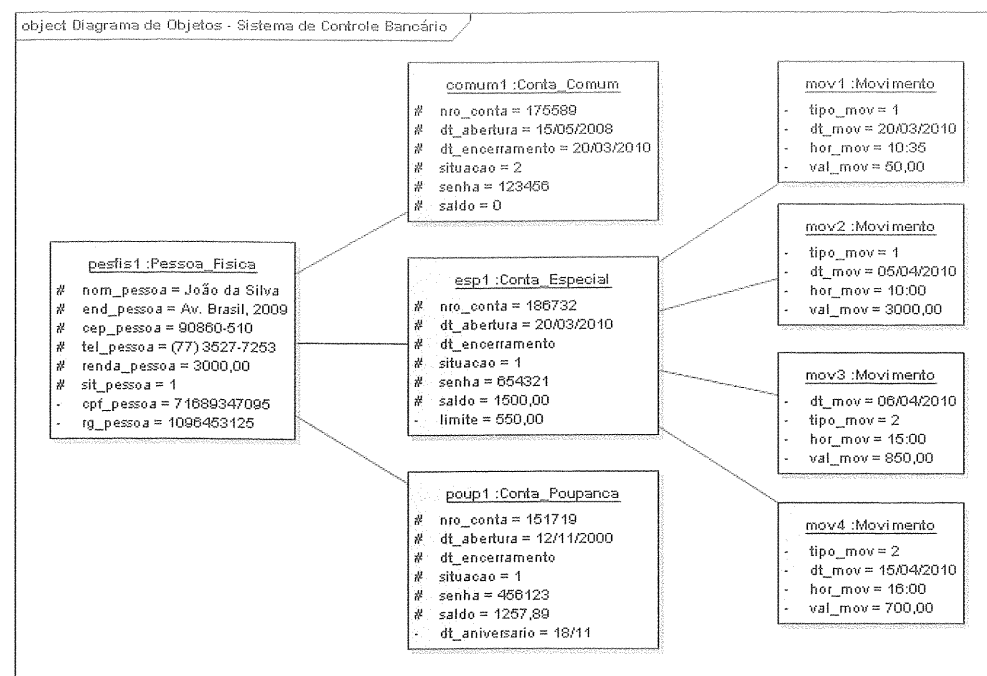


Figura 5.4 – Diagrama de Objetos.

Nesse exemplo percebe-se que o objeto `pesfis1`, da classe `Pessoa_Fisica`, está vinculado a três objetos: o primeiro da classe `Conta_Comum`, o segundo da classe `Conta_Especial` e o terceiro da classe `Conta_Poupanca`, chamados respectivamente de `comum1`, `esp1` e `poup1`. Dessa forma, podemos concluir que a pessoa física representada pelo objeto `pesfis1` possui ou possuiu três contas na instituição bancária. Ao examinarmos o objeto `comum1`, percebemos que essa conta já se encontra encerrada, uma vez que o atributo `situacao` armazena o valor 2, que, por convenção, significa que a conta está inativa, e porque o atributo `dt_encerramento` tem uma data definida. Já os outros dois objetos ainda estão ativos.

Se continuarmos examinando a figura perceberemos que o objeto `esp1` está vinculado a quatro objetos da classe `Movimento`, `mov1`, `mov2`, `mov3` e `mov4`. Os dois primeiros objetos referem-se a um depósito de valores, enquanto os dois últimos a saques.

Como podemos observar, nem todas as classes estão representadas por seus objetos nesse exemplo, o que é, aliás, recomendável. Um diagrama de objetos deve focar o menor conjunto possível de classes, porque as classes podem ter um número muito grande de objetos, e representar objetos de todas as classes em um diagrama tende a deixá-lo muito extenso e poluído. Assim, o melhor é criar vários diagramas de objetos enfocando pequenas partes do diagrama de classes.

CAPÍTULO 6

Diagrama de Pacotes

O diagrama de pacotes descreve como os elementos do modelo estão organizados em pacotes e demonstra as dependências entre eles. Esse diagrama é muito útil para a modelagem de subsistemas e para a modelagem de subdivisões da arquitetura de uma linguagem. Pode ser utilizado também para representar um conjunto de sistemas integrados, representados por pacotes, ou ainda os submódulos englobados por um sistema. O diagrama pode ser utilizado ainda para demonstrar a arquitetura de uma linguagem, como ocorre com a própria UML, ou a arquitetura de um processo de desenvolvimento.

6.1 Pacotes

Pacotes são utilizados para agrupar elementos e fornecer denominações para esses grupos. Um pacote pode representar um sistema, um subsistema, uma biblioteca ou uma etapa de um processo de desenvolvimento, entre outras alternativas. Um pacote pode inclusive conter outros pacotes. A figura 6.1 apresenta um exemplo de pacote.

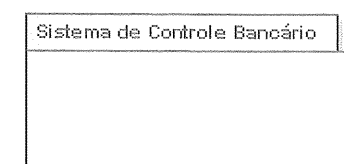


Figura 6.1 – Exemplo de Pacote.

O exemplo apresentado nessa figura contém um pacote intitulado `Sistema de Controle Bancário`, o que significa que ele engloba os elementos contidos nesse sistema. Esse exemplo apresenta somente o pacote, sem revelar seu conteúdo. No entanto, é possível encontrar pacotes com seu conteúdo ou parte dele explicitamente declarado, como demonstra a figura 6.2.

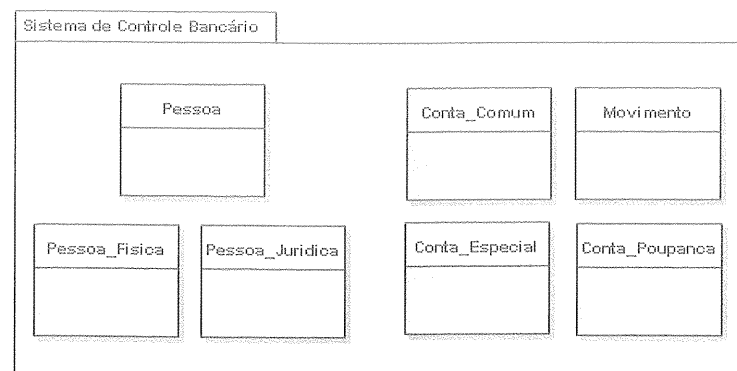


Figura 6.2 – Pacote com Detalhe de seus Membros.

Nesse exemplo, identificamos os elementos contidos pelo pacote, em termos de suas classes. Como o leitor pode notar, não definimos os atributos, métodos ou associações entre as classes. No entanto, isso é perfeitamente possível, podendo um pacote conter um diagrama de outro tipo completo dentro dele. O problema dessa abordagem é o grande espaço ocupado pelo pacote, sendo mais comum encontrar pacotes sem o detalhamento de seu conteúdo. Existe uma notação alternativa para identificar os membros do pacote por meio do conector de aninhamento, representado por um círculo contendo uma cruz, conforme demonstra a figura 6.3.

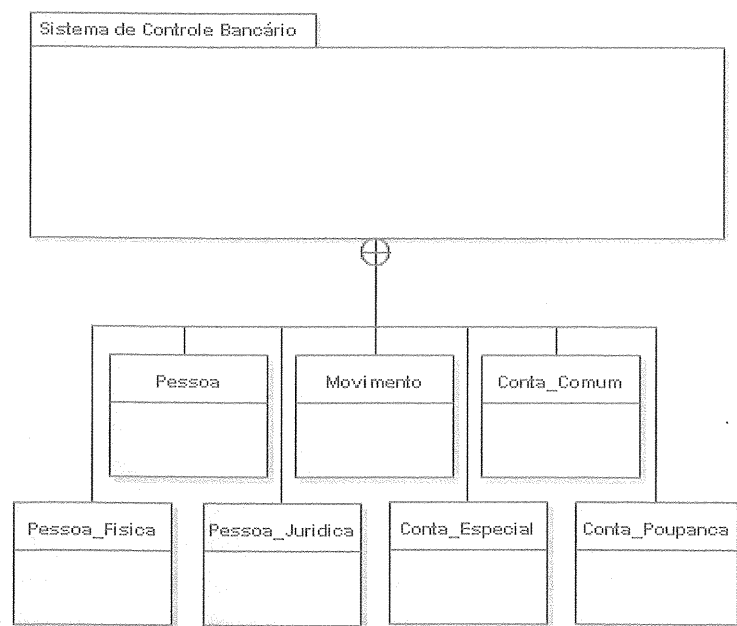


Figura 6.3 – Pacote com Detalhe de seus Membros – Notação Alternativa com Conector de Aninhamento.

6.2 Dependência

Pacotes normalmente contêm dependências entre si. Um relacionamento de dependência informa que o elemento dependente necessita de alguma forma do elemento do qual depende. A figura 6.4 apresenta um exemplo de relacionamento de dependência entre pacotes.

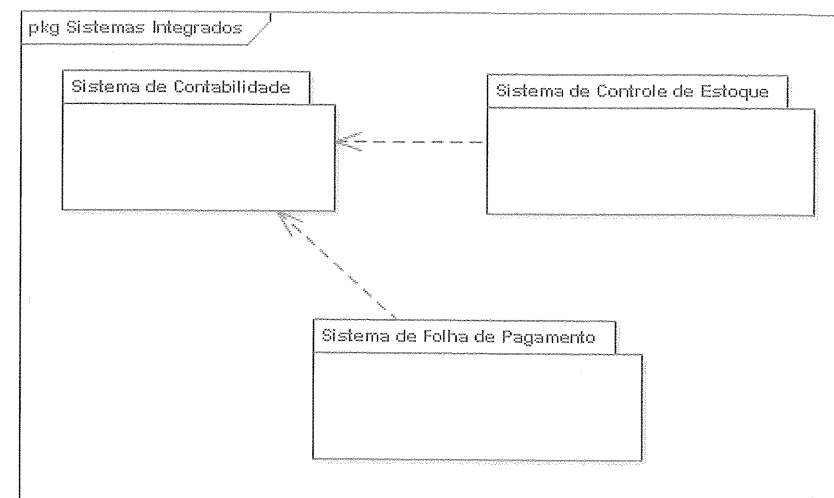


Figura 6.4 – Dependência entre Pacotes.

Nesse exemplo existem três sistemas integrados: o sistema de contabilidade, de estoque e de folha de pagamento. Os pacotes estão associados por meio de dependências, indicando que os sistemas de estoque e de folha de pagamento necessitam do sistema de contabilidade para lançar suas operações financeiras. Outro exemplo de dependência é apresentado na figura 6.5.

Nesse exemplo, os pacotes representam as etapas de um processo de desenvolvimento de software, onde primeiramente se define a interface, em seguida a aplicação propriamente dita e finalmente a forma como os dados serão persistidos fisicamente em disco.

O relacionamento de dependência no diagrama de pacotes pode ter dois estereótipos: `<<merge>>`, significando que os elementos do pacote que utiliza essa dependência serão unidos aos elementos do outro pacote, e `<<import>>`, significando que o pacote que utiliza essa dependência está importando alguma característica ou elemento do outro pacote.

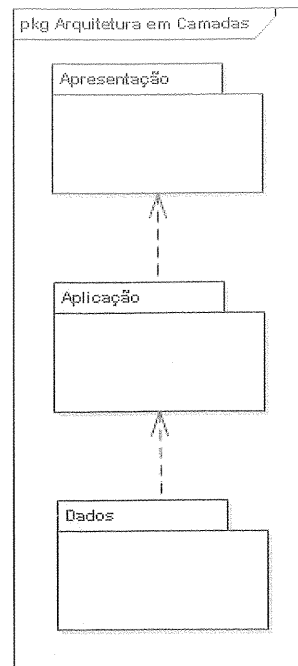


Figura 6.5 – Dependência entre Pacotes.

6.3 Pacotes Contendo Pacotes

É bastante comum, principalmente na definição de arquiteturas de linguagens de modelagem, que um pacote se subdivida em diversos outros pacotes. Isso ocorre, por exemplo, com a Biblioteca de Infraestrutura da UML, conforme a figura 6.6.

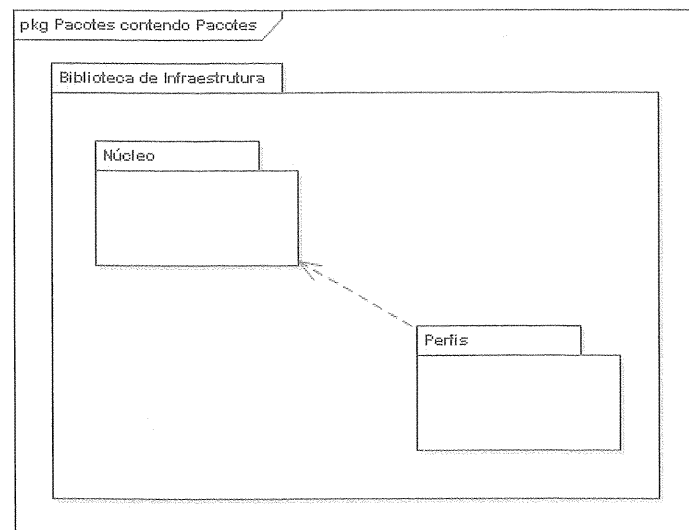


Figura 6.6 – Pacotes Contendo Pacotes.

Nesse exemplo percebemos que o pacote Biblioteca de Infraestrutura contém os pacotes Núcleo e Perfis e que o pacote Perfis tem uma relação de dependência com o pacote Núcleo.

6.4 Estereótipos Aplicados a Pacotes

É possível aplicar estereótipos aos pacotes, deixando claro que estes representam sistemas, subsistemas, frameworks ou modelos por exemplo, conforme apresentado na figura 6.7.

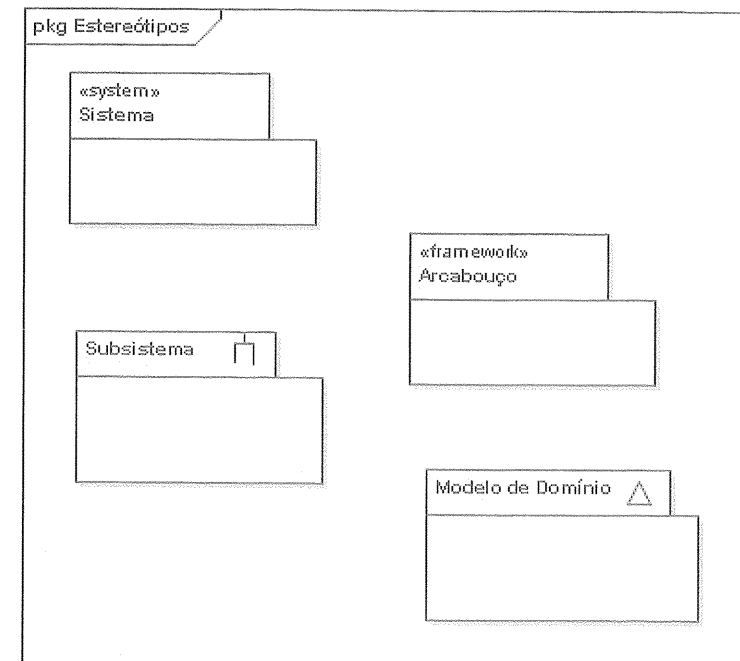


Figura 6.7 – Pacotes com Estereótipos.

Nessa figura apresentamos pacotes com estereótipos dos quatro tipos enunciados. Observe que enquanto os estereótipos `<<system>>` e `<<framework>>` são estereótipos de texto, `<<subsystem>>` e `<<model>>` são estereótipos gráficos que modificam um pouco o desenho-padrão do pacote.

CAPÍTULO 7

Diagrama de Sequência

Este é um diagrama comportamental que procura determinar a sequência de eventos que ocorrem em um determinado processo, identificando quais mensagens devem ser disparadas entre os elementos envolvidos e em que ordem. Assim, determinar a ordem em que os eventos ocorrem, as mensagens que são enviadas, os métodos que são chamados e como os objetos interagem dentro de um determinado processo é o objetivo principal desse diagrama.

O diagrama de sequência baseia-se no diagrama de casos de uso, havendo normalmente um diagrama de sequência para cada caso de uso declarado, uma vez que um caso de uso, em geral, refere-se a um processo disparado por um ator. Assim, um diagrama de sequência também permite documentar um caso de uso específico, sendo que muitas ferramentas CASE permitem gerar um diagrama de sequência diretamente a partir de um caso de uso.

Obviamente, o diagrama de sequência depende também do diagrama de classes, já que as classes dos objetos utilizados no diagrama de sequência estão descritas nele. No entanto, o diagrama de sequência é uma excelente forma de validar e complementar o diagrama de classes, pois é ao modelar um diagrama de sequência que se percebe quais métodos são necessários declarar em que classes. Isto é, inclusive, o que é recomendado fazer em alguns processos de desenvolvimento de software, como o Processo Unificado, no qual, como já foi dito anteriormente, primeiramente produz-se o modelo conceitual, durante a fase de análise, e só mais tarde, durante a fase de projeto, produz-se o modelo de domínio, onde serão detalhados os métodos das classes. A descoberta desses métodos é feita por meio do detalhamento dos processos enunciados no diagrama de casos de uso, por meio de diagramas de interação, como os de sequência, embora estes possam também ser utilizados na fase de análise, como será demonstrado ao longo deste capítulo.

7.1 Atores

Os atores modelados neste diagrama são instâncias dos atores declarados no diagrama de casos de uso, representando entidades externas que interagem com o sistema e que solicitam serviços, gerando, assim, eventos que iniciam processos. Esses atores costumam ser apresentados como bonecos magros idênticos aos usados no diagrama de casos de uso, porém, contendo uma linha de vida. O conceito de linha de vida será explicado nas seções seguintes. A figura 7.1 ilustra um exemplo de como é representado um ator no diagrama de sequência.

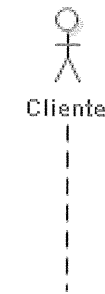


Figura 7.1 – Exemplo de Ator.

A figura 7.1 representa um cliente que interage com o sistema ou com outro ator envolvido no processo. Os atores não são realmente obrigatórios nesse diagrama, mas são utilizados com muita frequência. Além disso, como a maioria dos diagramas de sequência, senão todos, refletem o aspecto dinâmico de um caso de uso, a utilização dos mesmos atores que interagem com o caso de uso em questão, facilita a compreensão do processo.

7.2 Lifelines

Um lifeline é um participante individual em uma interação. Na maioria das vezes um lifeline irá se referir a uma instância de uma classe que participa de uma interação. Para não confundir com a linha que determina o tempo em que um participante existe no diagrama, iremos manter essa terminologia em inglês. Lifelines no diagrama de sequência têm a mesma notação utilizada no diagrama de objetos, diferenciando-se por uma linha de vida, representada por uma linha vertical tracejada abaixo do participante. Como na prática, em geral um lifeline é um objeto, iremos utilizar o termo objeto com frequência ao longo do capítulo. A figura 7.2 apresenta um exemplo de lifeline no diagrama de sequência.

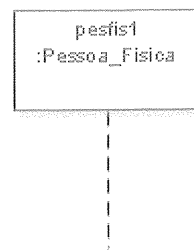


Figura 7.2 – Exemplo de Lifeline.

Como é possível perceber ao observarmos a figura 7.2, existe um lifeline chamado pesfis1, e esse lifeline é uma instância da classe Pessoa_Fisica. A linha tracejada vertical que surge a partir do objeto representa a linha de vida do mesmo.

Um lifeline pode existir desde o início do processo ou ser criado durante o decorrer da execução do mesmo. No primeiro caso, o retângulo que representa o lifeline aparecerá na parte superior do diagrama. Já no segundo caso, o retângulo surgirá na mesma altura em que a mensagem que o criar for chamada. A figura 7.3 apresenta um exemplo de lifelines ativos desde o início do processo e lifelines criados no decorrer do mesmo.

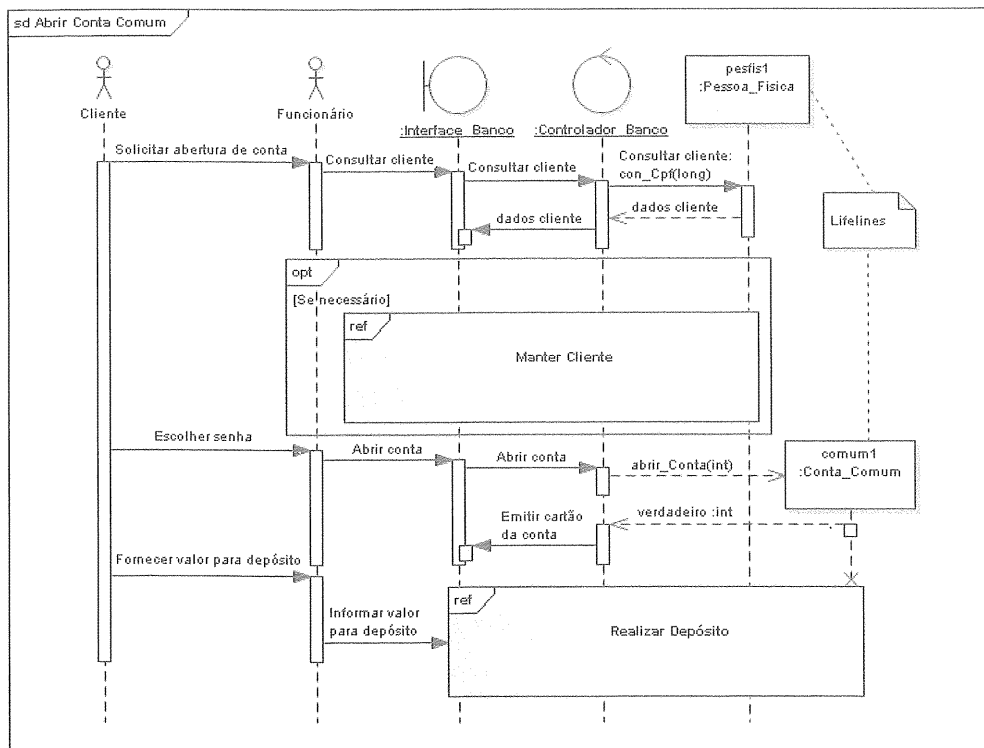


Figura 7.3 – Lifelines em um Diagrama de Sequência.

Na figura 7.3 utilizamos o componente notas para destacar a existência de dois lifelines no diagrama, pesfis1 e comum1, esse último pertencente à classe Conta_Comum. Ao estudarmos a figura verificamos que o objeto (lifeline) pesfis1 esteve ativo desde o início do processo. Já o objeto (lifeline) comum1 foi instanciado ao longo do processo. Observe que há mais duas instâncias de objetos no diagrama, cujos nomes não foram definidos, que se referem respectivamente à classe Interface_Banco, uma classe de fronteira e à classe Controlador_Banco, uma classe de controle. Além disso, existem dois atores interagindo nesse processo, os atores Cliente e Funcionário. Outros detalhes deste diagrama serão explanados ao longo deste capítulo.

7.3 Linha de Vida

A linha de vida representa o tempo em que um objeto (lifeline) existe durante um processo. As linhas de vida são representadas por linhas finas verticais tracejadas, partindo do retângulo que representa o objeto. A linha de vida é interrompida com um “X” quando o objeto é destruído. Um objeto não precisa necessariamente existir quando o processo é iniciado, podendo ser criado ao longo do mesmo. Assim, os objetos criados não são representados no topo do diagrama, mas só a partir do momento em que forem criados, como no exemplo da figura 7.3, e obviamente só terão uma linha de vida a partir desse mesmo momento.

7.4 Foco de Controle ou Ativação

Indica os períodos em que um determinado objeto está participando ativamente do processo, ou seja, identifica os momentos em que um objeto está executando um ou mais métodos utilizados em um processo específico. Os focos de controle são representados dentro da linha de vida de um objeto, porém, enquanto as linhas de vida são representadas por tracejados finos, o foco de controle é representado por uma linha mais grossa. A figura 7.4 apresenta um exemplo de linha de vida e foco de controle.

Nessa figura novamente utilizamos o componente notas, dessa vez para identificar a linha de vida e o foco de controle de um objeto do diagrama. Ao observarmos a figura 7.4, podemos perceber, por intermédio da linha de vida (linha vertical tracejada fina que parte do objeto), que o objeto pesfis1 esteve presente durante todo o processo de abertura de conta, mas ele só participou ativamente do processo quando do disparo do método conCpf, quando a linha de vida tornou-se mais grossa, indicando que o foco de controle do processo estava sobre o objeto pesfis1.

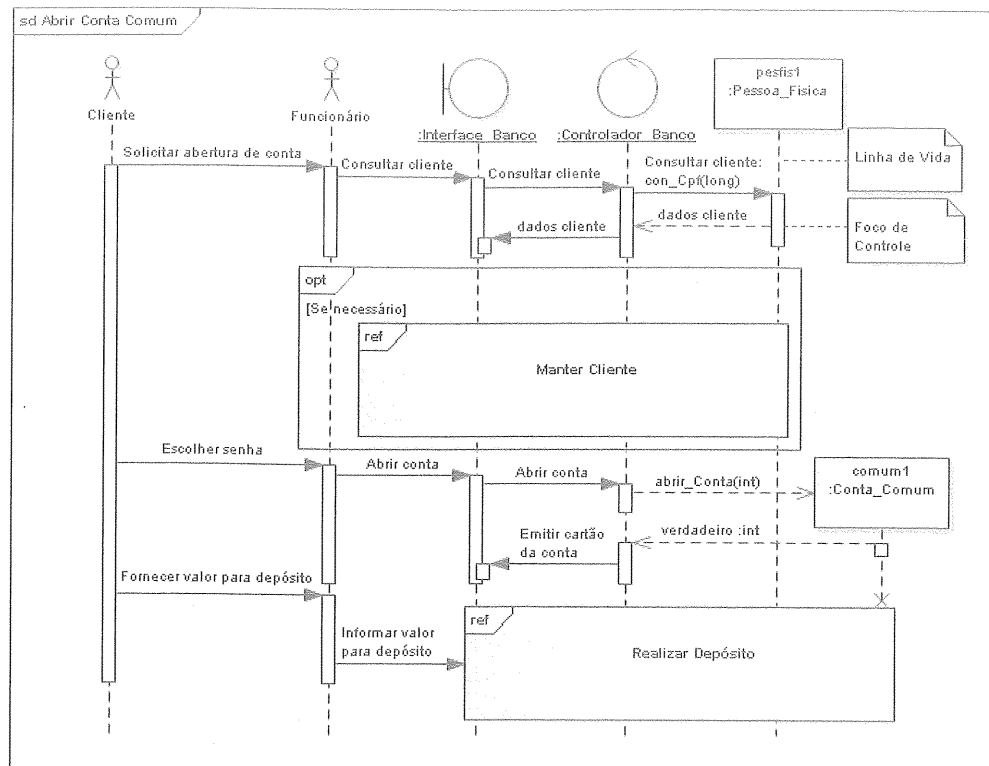


Figura 74 – Exemplo de Linha de Vida e Foco de Controle.

7.5 Mensagens ou Estímulos

As mensagens são utilizadas para demonstrar a ocorrência de eventos, que normalmente forçam a chamada de um método em algum dos objetos envolvidos no processo. Pode ocorrer, no entanto, de uma mensagem representar a comunicação entre dois atores, nesse caso, não disparando métodos. Um diagrama de sequência em geral é iniciado por um evento externo, causado por algum ator, o que acarreta o disparo de um método em um dos objetos. As mensagens podem ser disparadas entre:

- um ator e outro ator;
- um ator e um objeto, onde um ator produz um evento que dispara um método em um objeto;
- um objeto e outro objeto, o que constitui a ocorrência mais comum de mensagens, onde um objeto transmite uma mensagem para outro, em geral solicitando a execução de um método. Um objeto pode inclusive enviar uma mensagem para si mesmo, disparando um método em si próprio, o que é conhecido como autochamada;

- um objeto e um ator, o que normalmente ocorre quando um objeto envia uma mensagem de retorno em resposta à chamada de um método solicitado, contendo seus resultados.

As mensagens são representadas por linhas entre dois componentes, contendo setas indicando qual componente enviou a mensagem e qual a recebeu. As mensagens são apresentadas na posição horizontal entre as linhas de vida dos componentes e sua ordem sequencial é demonstrada de cima para baixo.

Os textos contidos nas mensagens primeiramente identificam qual evento ocorreu e forçou o envio da mensagem e qual método foi chamado. As duas informações são separadas por um símbolo de dois pontos (:). Podem ocorrer eventos que não disparam métodos. Nesse caso, a mensagem descreve apenas o evento que ocorreu, sem o símbolo de dois pontos e nenhum texto após os mesmos. Também pode acontecer de somente o método chamado ser descrito, sem detalhar qual evento o causou.

A figura 7.5 apresenta um exemplo de mensagem disparada entre atores, representando uma conversa entre os mesmos e que, portanto, não gera o disparo de nenhum método. Já a figura 7.6 mostra um exemplo de uma mensagem enviada por um objeto que dispara um método em outro.

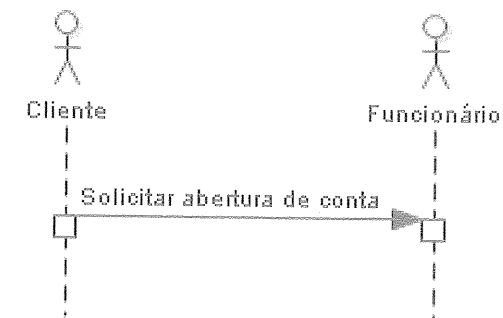


Figura 7.5 – Mensagem simples entre atores.

Ao compararmos as duas figuras, percebemos que a mensagem da figura 7.5 descreve simplesmente o evento em si, enquanto a da figura 7.6 descreve o evento e, após os dois pontos, o método que foi disparado por ele. Logicamente, tais métodos podem conter parâmetros e retornar valores. No entanto, deve-se evitar colocar muitos detalhes nas chamadas dos métodos para impedir que o diagrama de sequência torne-se muito extenso.

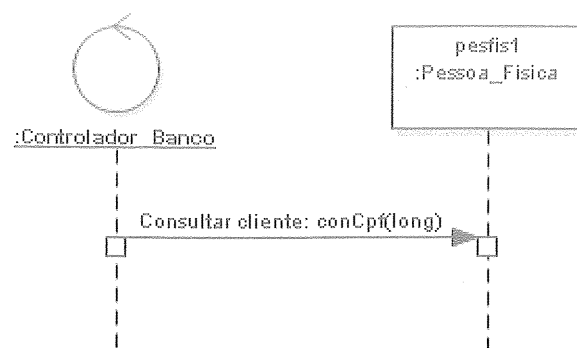


Figura 7.6 – Mensagem com disparo de método entre objetos.

Quando a mensagem é dirigida a um objeto já existente, a seta da mensagem atinge a linha de vida do objeto, engrossando-a, identificando que o foco de controle está sobre o objeto em questão. No entanto, quando a mensagem cria um novo objeto, a seta atinge o retângulo que representa o objeto, indicando que a mensagem representa um método construtor e que o objeto passa a existir somente a partir daquele momento. A figura 7.7 apresenta um exemplo de mensagem que provoca a criação de um novo objeto.

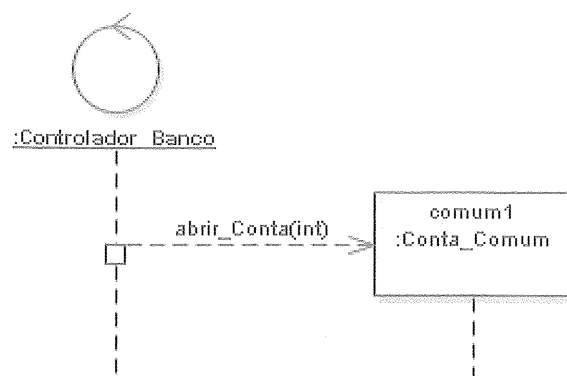


Figura 7.7 – Mensagem que instancia um novo objeto.

Nesse exemplo, o objeto da classe `Controlador_Banco` dispara o método `abrir_Conta` no objeto `comum1` da classe `Conta_Comum`, instanciando esse objeto a partir desse momento.

Uma mensagem pode também representar um método destrutor, ou seja, um método que elimina um objeto não mais necessário. Nesse caso a mensagem atinge a linha de vida de um objeto e a interrompe com um X. A figura 7.8 apresenta um exemplo de chamada de método destrutor.

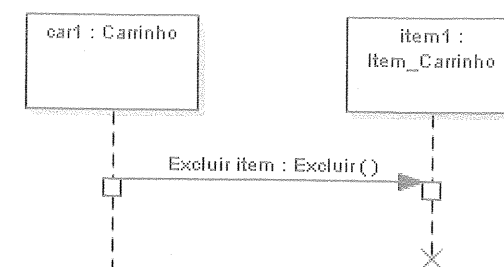


Figura 7.8 – Mensagem que dispara um método destrutor.

Nesse exemplo, existe um objeto `car1` pertencente a uma classe `Carrinho`, que representa um carrinho de compras, como os encontrados nas livrarias digitais da internet e que pode ter muitos itens, representados pelos objetos da classe `Item_Carrinho`. Se em algum momento o cliente resolver cancelar a compra de algum dos itens do carrinho, o objeto da classe `Carrinho` deverá disparar um método destrutor no objeto da classe `Item_Carrinho`, aqui representado pelo método `Excluir`.

7.6 Mensagens de retorno

Esse tipo de mensagem identifica a resposta a uma mensagem para o objeto ou ator que a chamou. Uma mensagem de retorno pode retornar informações específicas do método chamado ou apenas um valor indicando se o método foi executado com sucesso ou não. As mensagens de retorno são representadas por uma linha tracejada contendo uma seta fina que aponta para o objeto que recebe o resultado do método chamado. A figura 7.9 apresenta um exemplo de mensagem de retorno.

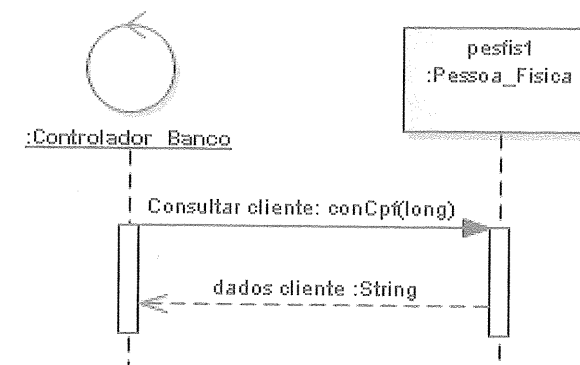


Figura 7.9 – Mensagem de retorno.

Aqui a mensagem de retorno emitida pelo objeto `pesfis1` para o objeto da classe `Controlador_Banco`, após este ter disparado o método `conCpf` no primeiro objeto, retorna os dados do cliente consultado. Observe que o valor retornado é do tipo `String`. Alguns autores só modelam as mensagens de retorno consideradas realmente importantes para evitar deixar o diagrama muito poluído.

7.7 Autochamadas ou Autodelegações

Autochamadas são mensagens que um objeto envia para si mesmo. No caso de autochamadas, uma mensagem parte da linha de vida do objeto e atinge a linha de vida do próprio objeto. A figura 7.10 demonstra um exemplo de autochamada.

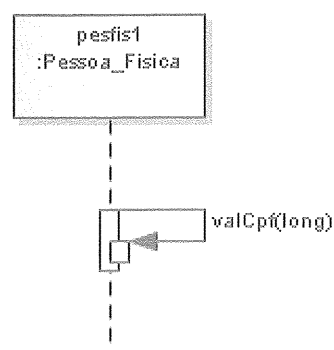


Figura 7.10 – Autochamada.

No exemplo ilustrado na figura 7.10, o objeto `pesfis1` dispara em si mesmo a chamada ao método de validação de CPF, `valCpf`.

7.8 Detalhes de Tempo

Às vezes pode ser necessário definir detalhes de tempo de uma mensagem, como por exemplo, o tempo máximo de espera até que uma mensagem seja disparada. Quando se quer demonstrar o tempo que uma mensagem leva em consideração antes de ser disparada, deve-se usar restrições de duração, e a mensagem, em vez de ser representada na horizontal, como é o padrão, é apresentada na diagonal, como demonstra o exemplo da figura 7.11.

Aqui enfocamos um sistema de venda pela internet, onde o controlador de vendas espera até 30 minutos o cliente realizar mais alguma operação sobre o pedido. Se esse tempo for ultrapassado sem haver qualquer movimentação ou confirmação no pedido, este será cancelado. Observe que a mensagem que dispara o método que cancelará o pedido tem uma restrição de duração que determina

que se devem esperar 30 minutos antes de disparar o método `cancelar_Pedido`, que é um método destrutor.

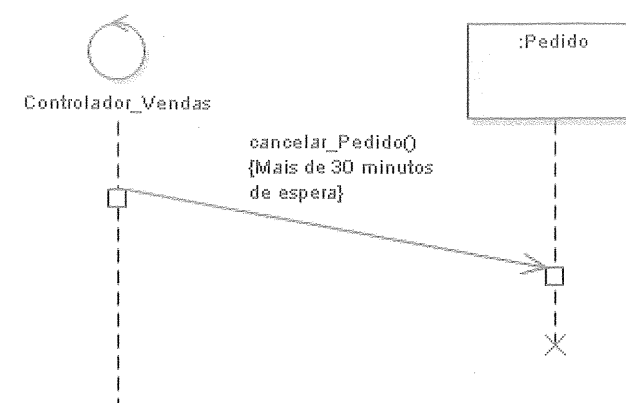


Figura 7.11 – Detalhes de Tempo.

7.9 Mensagens Perdidas e Mensagens Encontradas

Uma mensagem perdida representa uma mensagem que foi enviada e sua confirmação de recebimento não foi recebida, podendo significar que a mensagem não chegou ao seu destino, ou pode ainda representar uma mensagem enviada a um destino não representado no diagrama. Já uma mensagem encontrada representa o recebimento de uma mensagem enviada por um elemento desconhecido ou um elemento não representado no diagrama, ou o recebimento de uma mensagem que fora dada como perdida, pois seu tempo de espera por resposta poderia ter sido encerrado.

Tanto as mensagens perdidas como as mensagens encontradas são representadas por um círculo preenchido. Quando se trata de uma mensagem perdida, o círculo é atingido pela mensagem; já quando se trata de uma mensagem encontrada, esta parte do círculo. Uma aplicação para o uso de mensagens perdidas e mensagens encontradas pode ser a representação de troca de mensagens entre objetos localizados em máquinas (hosts) diferentes, possivelmente distantes entre si, onde a comunicação é realizada por meio de algum tipo de protocolo de rede. A figura 7.12 apresenta um exemplo de mensagem perdida e mensagem encontrada.

Aqui apresentamos um exemplo de mensagem perdida e outro de mensagem encontrada, onde uma mensagem foi enviada pelo objeto transmissor e, aparentemente, não foi recebida pelo objeto receptor. Após um tempo maior que o tempo máximo de espera, a mensagem de resposta foi recebida, ou seja, “encontrada”.

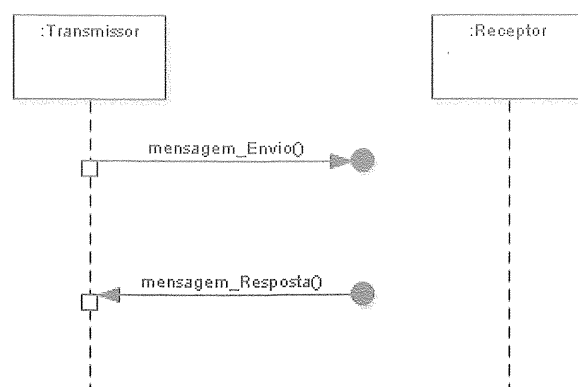


Figura 7.12 – Exemplo de Mensagem Perdida e Mensagem Encontrada.

7.10 Portas

O conceito de portas foi explicado no diagrama de classes. É possível representar um objeto no diagrama de sequência contendo instâncias das portas declaradas na classe a que ele pertence. Dessa forma o objeto poderá ter mais de uma linha de vida, o que permite a representação de mensagens externas e internas no objeto. A figura 7.13 apresenta um exemplo de instâncias de portas em objetos e o envio e recebimento de mensagens por elas.

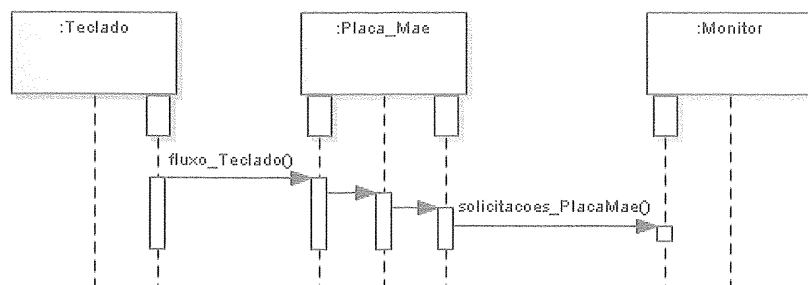


Figura 7.13 – Portas.

Nesse exemplo, instanciamos objetos das classes Teclado, Placa-Mae e Monitor. Se o leitor voltar ao capítulo 4 perceberá que essas classes se comunicam por meio de interfaces e portas. Aqui representamos o envio de fluxo de informações do teclado para a placa-mãe e, depois de estas terem sido processadas, a solicitação da placa-mãe ao monitor para que determinadas informações sejam apresentadas. Observe que as portas são representadas por retângulos abaixo do objeto, tendo suas próprias linhas de vida.

7.11 Fragmentos de Interação

Os fragmentos de interação são noções abstratas de unidades de interação geral. Um fragmento de interação é uma parte de uma interação. No entanto, cada fragmento de interação é considerado como uma interação independente. Um fragmento de interação é representado como um retângulo que envolve toda a interação, além de conter uma aba no canto superior esquerdo, contendo um operador que determina qual tipo de diagrama de interação ele se refere. O operador sd, por exemplo, indica que o fragmento é um diagrama de sequência ou comunicação. O texto seguinte ao operador contém a descrição da interação que está sendo modelada, normalmente contendo apenas o nome da interação. A figura 7.14 apresenta um exemplo de fragmento de interação.

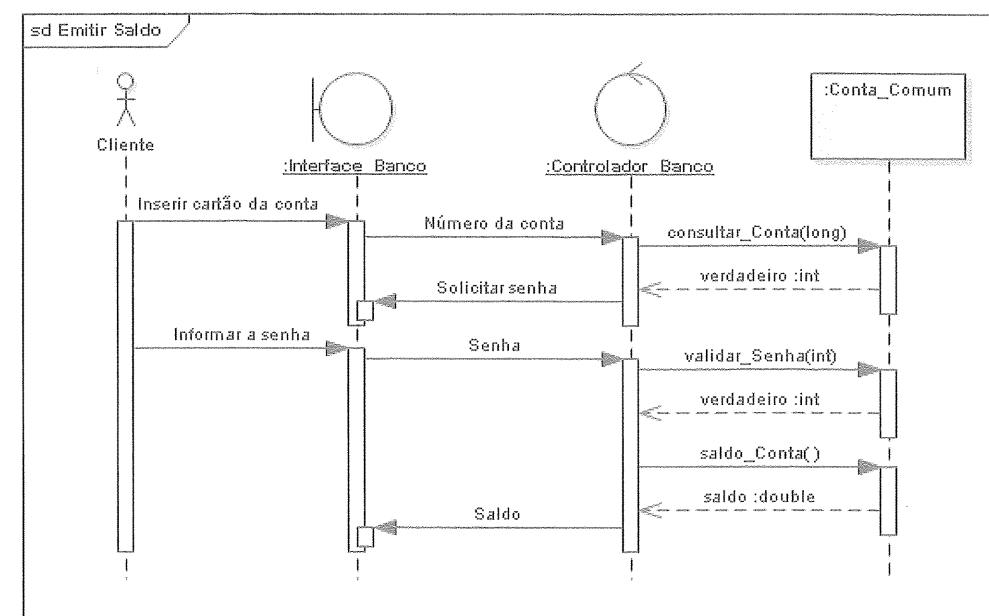


Figura 7.14 – Exemplo de Fragmento de Interação – Processo de Emissão de Saldo.

Essa figura representa o processo de emissão de saldo do sistema de controle bancário. Nesse processo, o cliente deve informar o número de sua conta, nesse caso inserindo o cartão da conta na interface do sistema (que pode ser física, como um caixa eletrônico). Esse número é repassado da interface para a controladora do sistema, que irá disparar o método consultar_Conta em um objeto da classe Conta_Comum e, caso o número da conta se refira a uma conta válida, o controlador solicitará à interface que apresente uma mensagem pedindo a senha da referida conta ao cliente.

O cliente então deverá digitar sua senha, que será repassada da interface ao controlador, que, por sua vez, solicitará o disparo do método `validar_Senha` e, se essa for a senha correta, o disparo do método `saldo_Conta` para retornar o valor do saldo da conta que o cliente informou. O controlador pedirá então a interface para apresentar o saldo ao cliente.

Observe que o diagrama demonstra o retorno “verdadeiro” para os primeiros dois métodos, destacando que se trata de um valor inteiro. Obviamente um valor inteiro não pode conter o texto “verdadeiro” e sim somente números, de modo que optamos por inserir a palavra “verdadeiro” para indicar o sucesso da operação, mas o retorno correto seria um número, sendo 1 para determinar que o cliente foi encontrado ou a senha informada é válida e 0 para o contrário, por exemplo.

7.12 Usos de Interação (Ocorrências de Interação antes da UML 2.1.1)

Uma das principais vantagens do uso de fragmentos de interação caracteriza-se pela possibilidade de se poder referenciá-los por meio do operador `Ref`, que é a abreviatura de `Referred` (referido) e significa que se deve procurar por um diagrama cujo nome é o mesmo do nome apresentado após o operador `Ref`, ou seja, o fragmento faz referência a outro diagrama, não detalhado no diagrama em questão e que deve ser inserido neste. A isto chama-se uso de interação (esse recurso passou a chamar-se uso de interação a partir da UML 2.1.1, antes era chamado ocorrência de interação), e permite que se montem diagramas mais complexos que fazem referência a outros diagramas como se fossem sub-rotinas, detalhadas em separado, diminuindo assim o tamanho do diagrama e facilitando sua leitura e compreensão. A figura 7.15 apresenta um exemplo de uso de interação em um fragmento de interação.

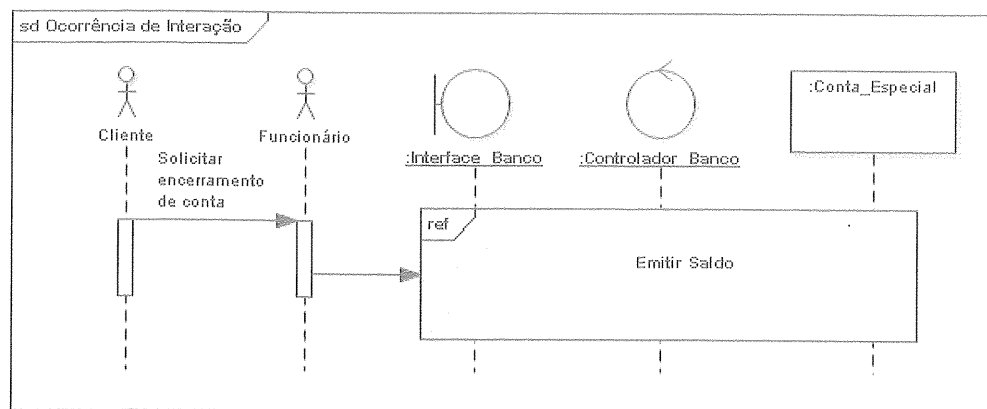


Figura 7.15 – Exemplo de Ocorrência de Interação.

Nesse exemplo enfocamos o início do processo de encerramento de conta do sistema de controle bancário, onde o cliente solicita ao funcionário o encerramento de uma conta. Para encerrar uma conta é necessário primeiro verificar seu saldo, para determinar se é preciso sacar ou depositar algum valor.

Como o processo de emissão de saldo já se encontra modelado em um diagrama à parte, é contraproducente ter que modelar esses passos novamente, além do que, se houver alguma mudança no processo, ele terá que ser alterado nos dois diagramas. Assim, faz-se uma referência a esse diagrama por meio de um uso de interação. Observe que o uso de interação é colocado sobre as linhas de vida dos objetos envolvidos no processo, e o ator (poderia ser também um objeto) simplesmente solicita sua execução por meio de uma mensagem. Essa mensagem poderia conter um texto, mas o próprio uso de interação já é autoexplicativo.

Assim, no processo de encerramento de conta, primeiramente o funcionário chamará o processo de emissão de saldo, que está detalhado em outro diagrama, e após a execução deste, o processo de encerramento de conta continuará normalmente.

É possível encontrar usos de interação simplesmente sobrepostos às linhas de vida dos objetos que fazem parte do processo, sem nem ao menos chamá-las por meio de uma mensagem, como se as instruções contidas nos usos de interação fossem adicionadas automaticamente ao diagrama, como pode ser visto na figura 7.16, que se refere ao processo de realizar saque.

Dessa vez, apresentamos o processo de realizar saque do sistema de controle bancário para ilustrar esse novo exemplo. Ao estudarmos o diagrama, percebemos que, da mesma forma que para visualizar o saldo de uma conta, é necessário primeiro informar o número da conta e, se esta estiver correta, informar a senha da mesma, para que se possa realizar um saque. Se a senha estiver correta, o sistema solicitará o valor que o cliente deseja sacar. O cliente então informará o valor desejado na interface, que o repassará para o controlador e este, por sua vez, disparará o método `sacar_valor` em um objeto da classe `Conta_Comum`. Esse método, caso haja saldo suficiente, diminuirá o valor solicitado do saldo da conta e autorizará o sistema a liberar o dinheiro pedido.

Conforme já foi definido no diagrama de casos de uso, ao realizar um saque é necessário registrar esse movimento e, como os passos do processo de registrar movimento estão detalhados em outro diagrama de sequência, apenas colocamos seu uso de interação sobre as linhas de vida dos objetos que participarão do processo, sem ao menos enviar uma mensagem a esse uso de interação.

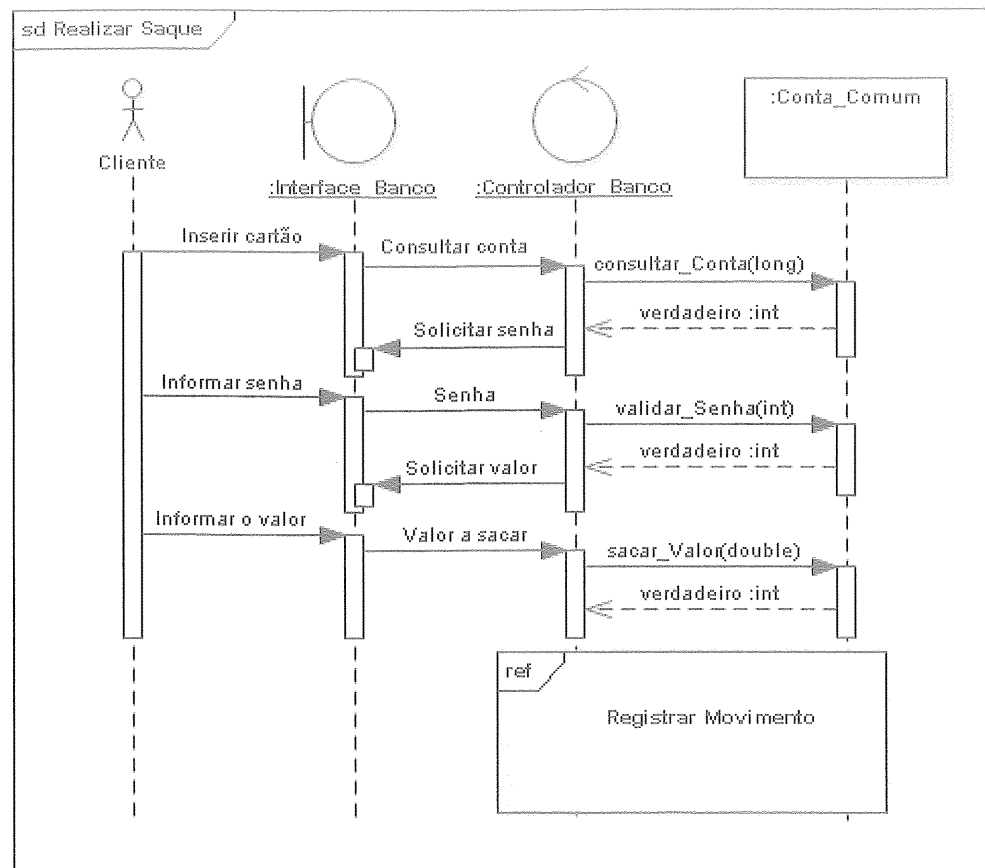


Figura 7.16 – Exemplo de Uso de Interação – Processo de Realizar Saque.

Os usos de interação podem se constituir em uma simples chamada a outro fragmento de interação ou podem passar parâmetros para o mesmo, receber o retorno da chamada deste ou ambos. Muitas vezes as associações de inclusão e extensão do diagrama de casos de uso denotam a necessidade da existência de usos de interação nos diagramas de sequência, já que um diagrama de sequência é uma forma de documentar um caso de uso e, se este tem uma associação de inclusão ou extensão com outro caso de uso, é muitas vezes preciso referenciá-la no diagrama de sequência, por meio de usos de interação.

7.13 Portões (Gates)

Um portão é uma interface entre fragmentos, um ponto de conexão para relacionar uma mensagem fora de um uso de interação com uma mensagem dentro do uso de interação. Portões podem ser representados de duas formas. A mais comum representa o portão simplesmente pelo encontro da seta da mensagem

no retângulo do uso de interação, como ocorre na figura 7.15. Quando, porém, se deseja explicitamente representar uma mensagem transmitida por algum elemento externo ou envio de uma mensagem para fora de um fragmento, o portão é representado por um pequeno quadrado, podendo este ser atingido por uma mensagem ou uma mensagem ser enviada a partir dele.

7.14 Fragmentos Combinados e Operadores de Interação

Nas versões anteriores à versão 2.0 da UML, os diagramas de sequência tinham dificuldade em trabalhar questões como testes se-senão, laços ou processamentos paralelos. Essas questões foram abordadas a partir da versão 2.0 por meio do uso de fragmentos combinados, que permitem uma modelagem semi-independente da parte do diagrama onde deve-se focar problemas como os enunciados.

Os fragmentos combinados são representados por um retângulo que determina a área de abrangência do fragmento no diagrama, além de conterem ainda uma subdivisão em sua extremidade superior esquerda para identificar a descrição do fragmento combinado e seu operador de interação, que define o tipo de fragmento que está sendo modelado. Alguns dos operadores de interação mais comuns são listados e exemplificados a seguir:

- **Alt** – Abreviatura de Alternatives (Alternativas). Este operador de interação define que o fragmento combinado representa uma escolha entre dois ou mais comportamentos. Esse tipo de fragmento combinado costuma utilizar condições de guarda (texto entre colchetes que estabelece uma regra ou condição), também conhecidas como restrições de interação, para definir o teste a ser considerado na escolha de um dos comportamentos. A figura 7.17 apresenta um exemplo de fragmento combinado com o operador **alt**. Aqui damos continuidade ao processo de encerramento de conta, onde, depois de verificar o saldo da conta, deverá ser feita uma escolha entre duas operações. Se o saldo da conta for positivo, ele executará um saque e entregará ao cliente o valor depositado. Já se o saldo estiver negativo, o cliente deverá depositar o valor necessário para cobrir o saldo negativo da conta antes de encerrá-la. Observe que em cada uma das alternativas foi feita uma referência a um uso de interação, na primeira relativa ao processo de realizar saque e na segunda ao processo de realizar depósito. Uma vez que esses processos já se encontram modelados em outros diagramas é mais prático somente referenciá-los.

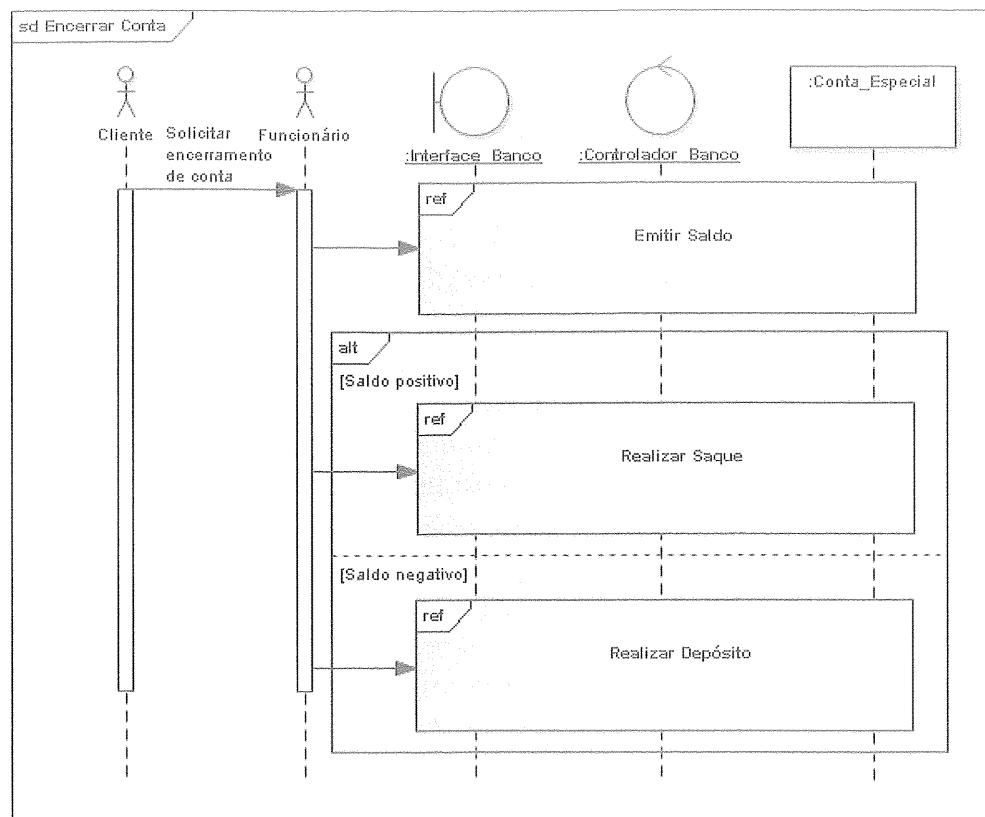


Figura 7.17 – Exemplo de Fragmento Combinado com Operador Alt.

Observe que fragmentos combinados que utilizam o operador de interação **alt** têm ao menos uma divisão, representada por uma linha tracejada, separando as ações executadas nas alternativas. Cada uma dessas divisões é chamada separador de operando de interação, e o conteúdo representado em cada divisão é conhecido como operando de interação, ou seja, uma área de atuação de um fragmento combinado. Um fragmento combinado contém ao menos um operando de interação, sendo que em alguns casos ele deve ter ao menos dois, como nesse exemplo, e em outros não poderá ter mais do que um, quando não existem fluxos alternativos ou paralelos. Nesse último caso, o operando de interação representa todo o conteúdo do fragmento combinado.

- **Opt** – Abreviatura de Option (Opção). Esse operador de interação determina que o fragmento combinado representa uma escolha de comportamento onde esse comportamento será ou não executado, não havendo uma escolha entre mais de um comportamento possível. A figura 7.18 apresenta um exemplo de fragmento combinado utilizando o operador de interação **opt**.

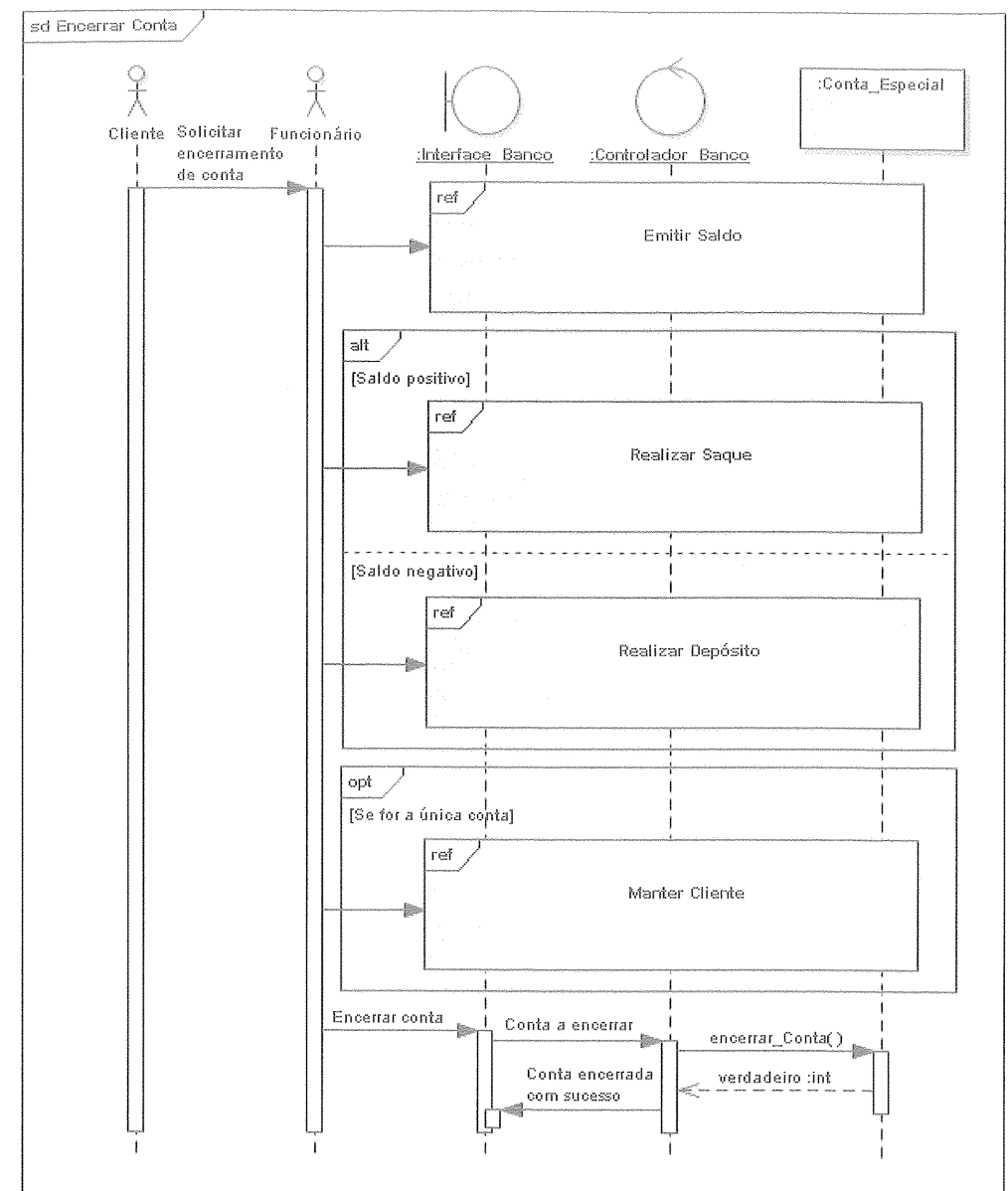


Figura 7.18 – Exemplo de Fragmento Combinado com Operador Opt.

A figura 7.18 dá continuidade ao processo de encerramento de conta, onde, após ser realizado um saque ou depósito, pode ser necessário dar manutenção no cadastro do cliente, tornando-o inativo, caso a conta a ser encerrada seja a única por ele possuída. Por esse motivo, utilizamos um fragmento combinado com operador **Opt**, significando que os passos nele contidos serão ou não executados dependendo de sua condição, determinada pela restrição de interação “Se for a única conta”.

O leitor notará que o processo *Manter Cliente* é um processo referenciado, ou seja, um uso de interação, e, se formos examinar o diagrama de casos de uso do sistema de controle bancário, perceberemos que o caso de uso *Encerrar Conta* tem uma associação de extensão com o processo *Manter Cliente*. No caso de associações de extensão é preciso fazer um teste para determinar se o caso de uso será ou não estendido, aqui refletido na restrição de interação. Quando se tratar de associações de inclusão não é preciso haver um fragmento combinado do tipo *opt*, bastando inserir um uso de interação sobre a linha de vida dos objetos, como acontece com o uso de interação *Emitir Saldo*.

Concluindo a descrição do processo de encerramento de conta, depois de verificar se é necessário alterar o cadastro do cliente, o funcionário solicitará o encerramento da conta em questão. Essa solicitação será repassada ao controlador, que irá disparar o método *encerrar_Conta* e, se o método for bem-sucedido, ordenará à interface que apresente uma mensagem de sucesso.

- **Par** – Abreviatura de Parallel (Paralelo). Esse operador de interação determina que o fragmento combinado representa uma execução paralela de dois ou mais comportamentos. A figura 7.19 apresenta um exemplo de fragmento combinado utilizando o operador de interação *par*.

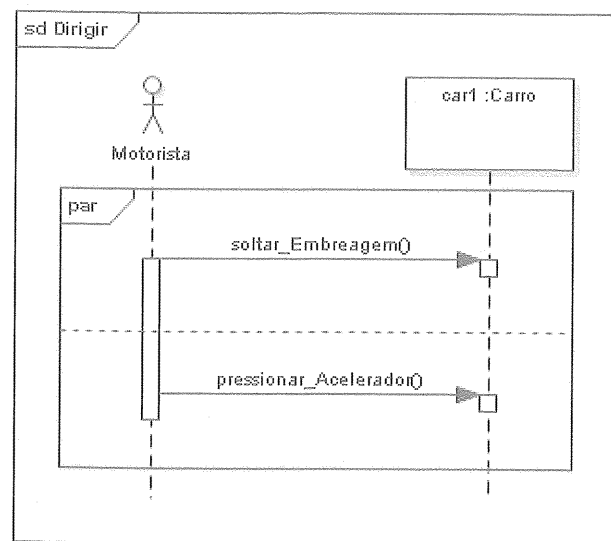


Figura 7.19 – Exemplo de Fragmento Combinado com Operador Par.

Identificamos aqui uma situação onde o ator *Motorista* deve realizar duas operações simultâneas sobre o objeto *car1* da classe *Carro* para poder dirigi-lo: soltar a embreagem e pressionar o acelerador. Note que uma linha tracejada divide os operandos de interação representando cada operação paralela.

- **Loop** Abreviatura de Looping (Laço). Esse operador de interação determina que o fragmento combinado representa um laço que poderá ser repetido diversas vezes. A figura 7.20 demonstra um exemplo de fragmento combinado utilizando o operador de interação *loop*.

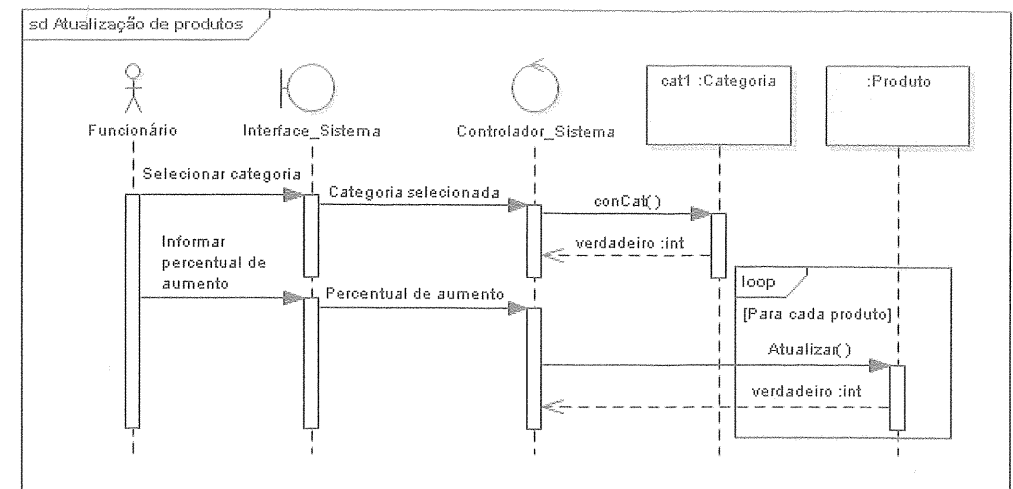


Figura 7.20 – Exemplo de Fragmento Combinado com Operador Loop.

Nesse exemplo identificamos um processo utilizado para aumentar os produtos de uma categoria. Ao observarmos a figura, percebemos que o funcionário seleciona uma categoria, informa o percentual de aumento e manda aumentar o valor de todos os produtos pertencentes à categoria selecionada. Percebemos ainda que o mesmo método é aplicado a cada produto pertencente à categoria selecionada, conforme demonstra a restrição de interação “[Para cada produto]”.

- **Break** (Quebra). Esse operador de interação indica uma “quebra” na execução normal do processo. É usado principalmente para modelar o tratamento de exceções. A figura 7.21 apresenta um exemplo de fragmento combinado utilizando o operador de interação *break*.

Aqui demos continuidade ao processo do exemplo anterior, onde identificamos uma situação em que pode ocorrer uma exceção, devido a algum registro estar corrompido ou algum campo ter valores inválidos. Assim criamos um método *tratExcecao* (Tratamento de Exceção), pertencente à classe *Exceção*, para tratar possíveis exceções que venham ocorrer durante a atualização. Observe ainda que, como uma exceção interrompe o desenrolar normal do laço, além de não ocorrer normalmente, ela está contida em um fragmento combinado do tipo *break*. Esse exemplo ilustra também a possibilidade de um fragmento combinado poder conter outro fragmento combinado.

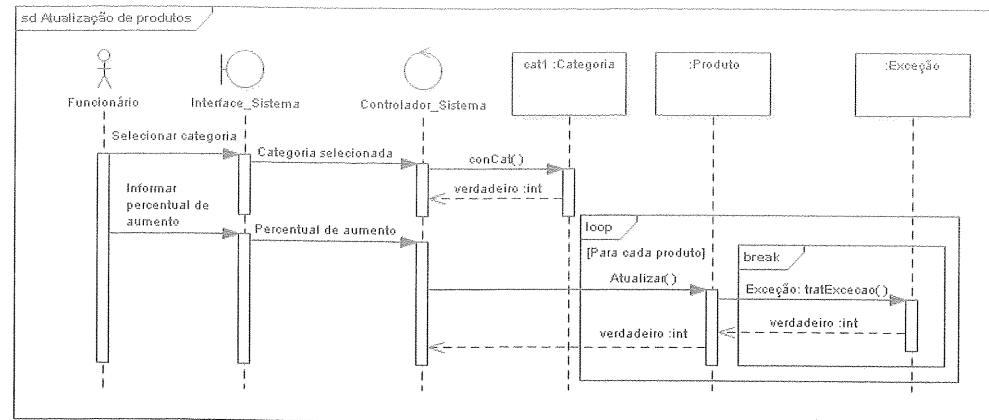


Figura 7.21 – Exemplo de Fragmento Combinado com Operador Break

- **Critical Region (Região Crítica)** – esse operador de interação identifica uma operação atômica que não pode ser interrompida por outro processo até ser totalmente concluída. A figura 7.22 apresenta um exemplo de fragmento combinado utilizando esse operador.

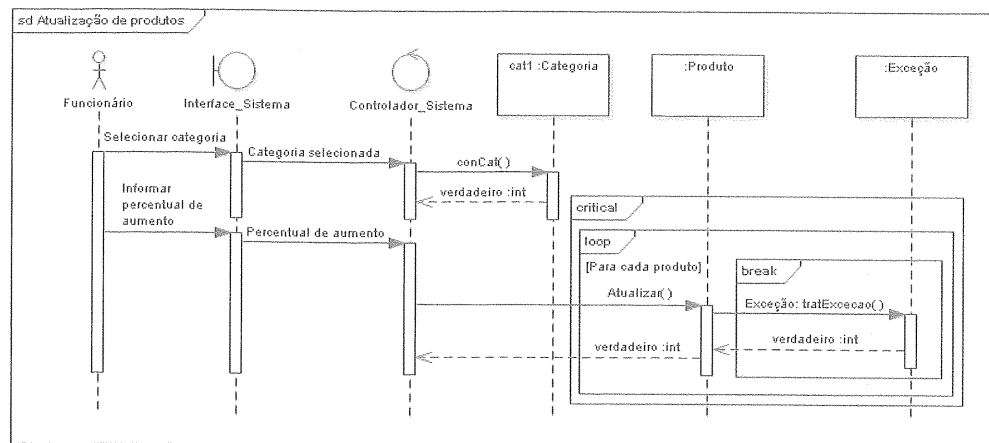


Figura 7.22 – Exemplo de Fragmento Combinado com Operador Critical Region.

Na figura 7.22 melhoramos o exemplo utilizado para ilustrar os dois últimos operadores de interação, envolvendo o laço de atualização de produtos com um fragmento combinado do tipo **critical**, indicando que a atualização dos produtos não deverá ser interrompida até que o processo seja totalmente concluído.

Existem ainda alguns operadores de interação menos utilizados, apresentados a seguir:

- **Neg** – Abreviatura de Negative (Negativo) – esse operador de interação representa eventos considerados como inválidos, que não devem ocorrer.

Todos os eventos não contidos em um fragmento combinado do tipo **neg** (quando existir um) são considerados positivos.

- **Assertion (Afirmção)** – esse operador de interação é o oposto do anterior, representando eventos considerados como válidos. Todos os eventos não contidos em um fragmento combinado do tipo **assertion** são automaticamente considerados negativos.
- **Ignore (Ignorar)** – o operador de interação Ignore determina que as mensagens contidas no fragmento devem ser ignoradas. Essas mensagens podem ser consideradas insignificantes e são intuitivamente ignoradas se aparecerem em uma execução correspondente. Alternativamente pode-se entender **ignore** como significando que as mensagens que são ignoradas podem aparecer em qualquer lugar nos eventos.
- **Consider (Considerar)** – esse operador de interação é o oposto do anterior e determina que as mensagens devem obrigatoriamente ser consideradas e que todas as outras mensagens não contidas no fragmento devem ser automaticamente desconsideradas. Tanto o operador de interação **ignore** como **consider** são frequentemente utilizados junto com os operadores **neg** e **assertion**, de maneira que um fragmento pode conter o outro.
- **Seq** – Abreviatura de Weak Sequencing (Sequência Fraca) – esse operador de interação identifica uma situação na qual as ocorrências de evento devem atender essas propriedades:
 1. As ordenações das ocorrências de evento dentro de cada um dos operandos são mantidas no resultado.
 2. Ocorrências de evento em linhas de vida diferentes de operandos diferentes podem vir em qualquer ordem.
 3. Ocorrências de evento na mesma linha de vida de operandos diferentes são ordenadas de tal forma que uma ocorrência de evento do primeiro operando venha antes do segundo operando.
- **Strict** – Abreviatura de Strict Sequencing (Sequência Estrita) – o operador de interação **strict** apresenta um refinamento do operador **weak sequencing** e garante que todas as mensagens no fragmento combinado sejam ordenadas do início ao fim.

7.15 Invariante de Estado (StateInvariant)

Um invariante de estado é uma restrição de tempo de execução aplicada aos participantes da interação. Ela pode ser usada para especificar diferentes tipos de restrições, tais como valores de atributos ou variáveis, estados internos ou

externos etc. Um invariante de estado é um fragmento de interação e deve ser colocado sobre uma linha de vida. A restrição de invariante de estado estabelece uma condição para que o comportamento pretendido de um ou mais elementos do diagrama possa ser executado. Essa condição deve ser avaliada durante o tempo de execução. Um invariante de estado pode ser representado por um círculo posicionado sobre a linha de vida do elemento ou por um texto entre chaves. A figura 7.23 apresenta um exemplo de invariante de estado.

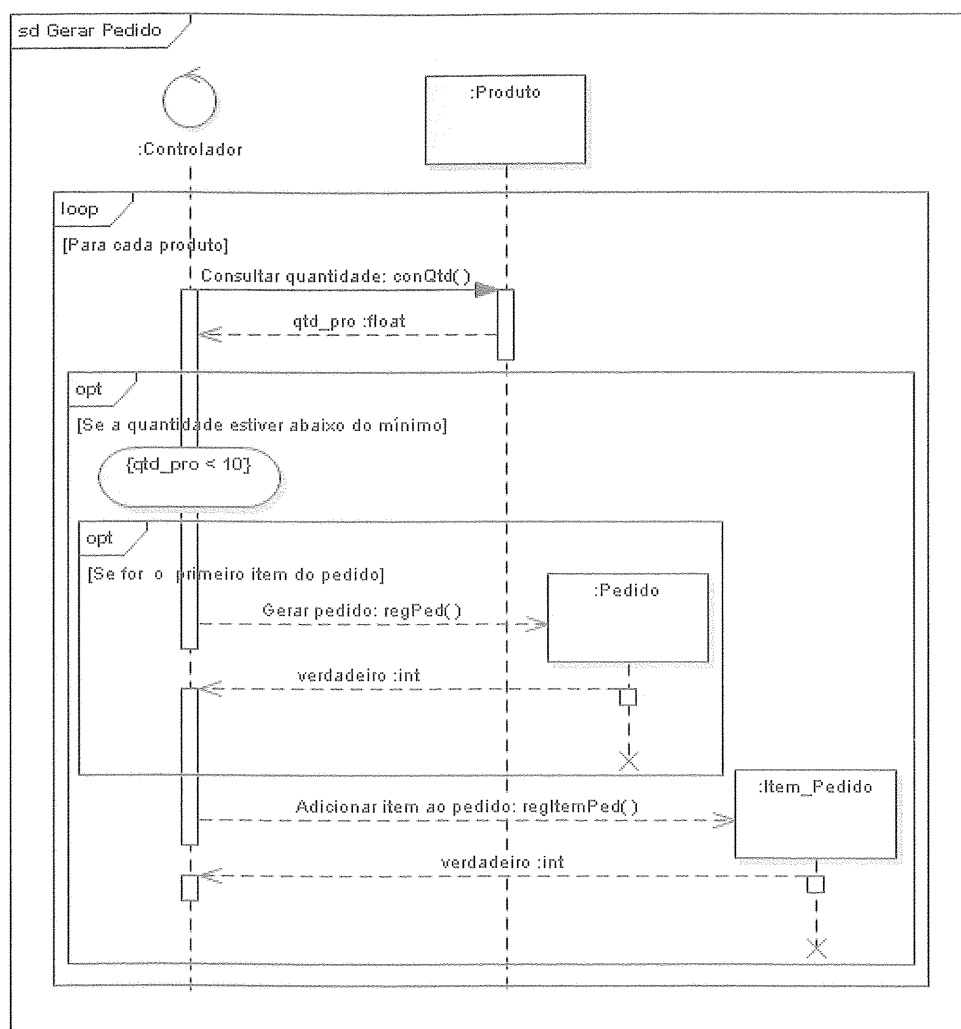


Figura 7.23 – Exemplo de Invariante de Estado.

Nesse exemplo, demonstramos uma situação em que é gerado um pedido automático para todos os produtos cuja quantidade está abaixo do mínimo. Há quatro lifelines representando as instâncias das classes Controlador, Produto, Pedido e

Item_Pedido. Há um fragmento combinado do tipo loop que representa um laço em que a controladora consultará a quantidade de cada produto. Ao receber a quantidade do produto, a controladora verifica se essa quantidade está abaixo do mínimo, isso é representado por um fragmento combinado do tipo opt e pela invariante de estado, que demonstra o teste que verifica se a quantidade é menor que dez. Em caso positivo, faz-se um segundo teste, representado por um segundo fragmento combinado do tipo opt que determina se o produto com quantidade abaixo do mínimo é o primeiro item do pedido. Nesse caso, é gerado um novo objeto da classe Pedido. Sempre que a quantidade de um produto estiver abaixo do mínimo, é gerado um novo objeto da classe Item_Pedido, representando um novo item do pedido.

7.16 Exemplos de Diagramas de Sequência

Nesta seção exemplificaremos os principais diagramas de sequência referentes ao sistema de controle bancário. Os processos de Emitir Saldo, Encerrar Conta e Realizar Saque já foram previamente explicados. Para ter uma compreensão completa desses diagramas é útil acompanhá-los junto com o diagrama de casos de uso e o diagrama de classes desse mesmo sistema, já explicados anteriormente.

7.16.1 Exemplo de Diagrama de Sequência – Abrir Conta Comum – Modelo Preliminar

O diagrama de sequência pode ser aplicado ainda durante a fase de análise, para se ter uma ideia geral de como será a interação de um processo. Contudo, durante a fase de análise os métodos não costumam estar definidos ainda, dessa forma o diagrama de sequência deve ser tratado como uma caixa preta, sem especificar como o processo se desenrolará internamente, identificando-se somente os atores envolvidos no processo, a interface do sistema e eventualmente a classe controladora, conforme pode ser visto na figura 7.24.

Aqui enfocamos o processo de abertura de conta comum, mas somente é apresentada a interação entre os usuários, a interface e a controladora, não havendo nenhum objeto de classe de entidade declarado, tampouco qualquer chamada de método. Esse diagrama apresenta apenas o pedido de execução de tarefas ao sistema e o retorno dos resultados das mesmas, não havendo detalhamento de como os serviços solicitados pelos atores serão realizados pelo sistema. Assim, quando o funcionário solicitar a consulta de um cliente, só será demonstrado que a controladora pedirá à interface para apresentar os dados do cliente consultado, sem definir como esses dados foram recuperados. O detalhamento do processo deve ser inserido somente na fase de projeto. Na seção seguinte apresentaremos esse mesmo diagrama, já identificando os objetos de classes de entidade envolvidos no processo e os métodos disparados entre eles.

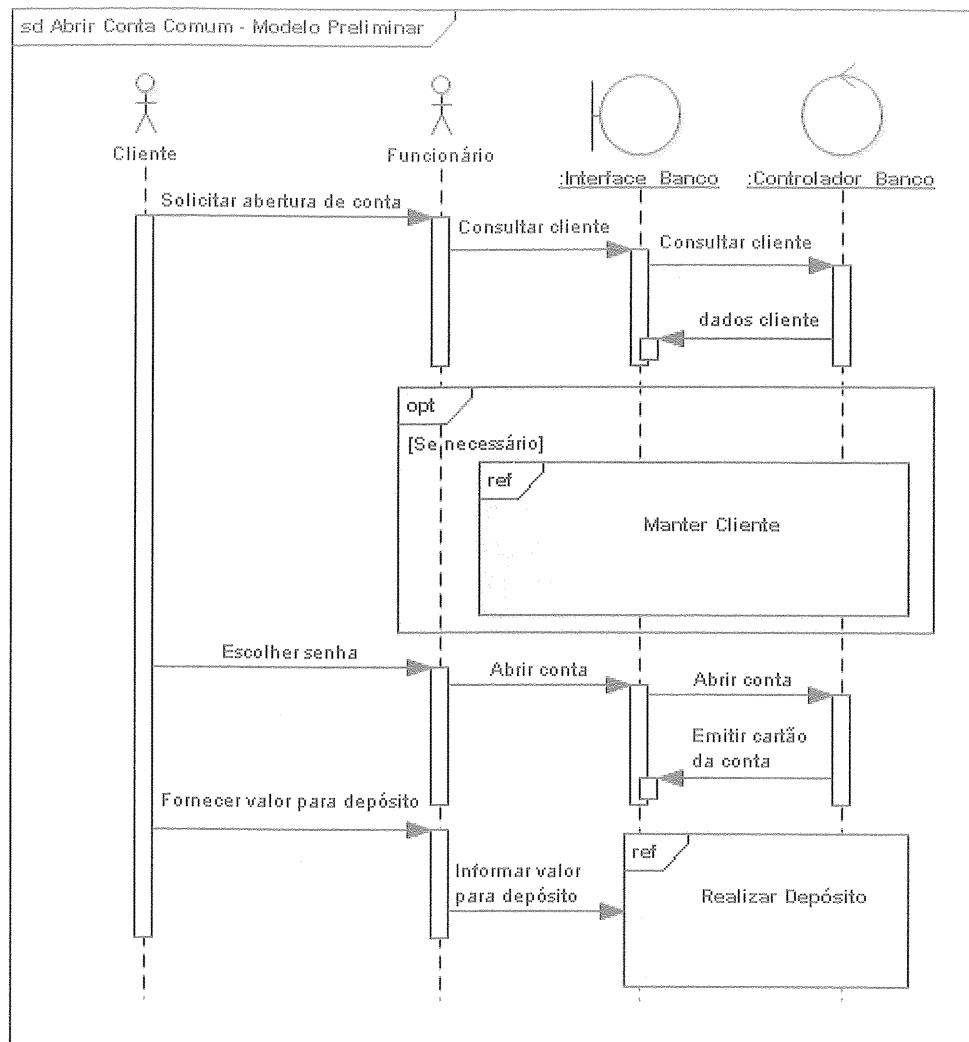


Figura 7.24 – Processo de Abertura de Conta Comum – Modelo Preliminar.

7.16.2 Exemplo de Diagrama de Sequência – Abrir Conta Comum – Modelo Detalhado

Neste exemplo damos continuidade ao processo representado pelo caso de uso **Abrir Conta Comum**, iniciado na seção anterior. Desta vez apresentamos o processo completo. Esse processo envolve os atores **Cliente** e **Funcionário**, uma instância (lifeline) da classe de fronteira **Interface_Banco**, uma instância da classe de controle **Controlador_Banco** e instâncias das classes de entidade **Pessoa_Fisica** e **Conta_Comum**. A figura 7.25 apresenta o diagrama de sequência referente a este processo.

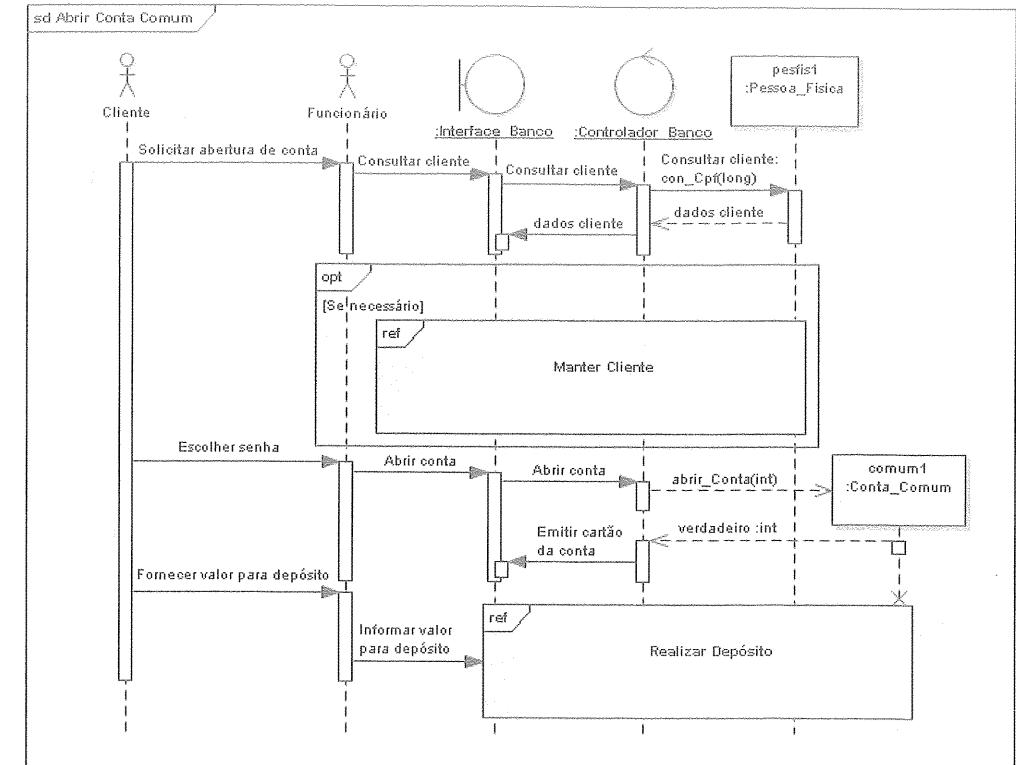


Figura 7.25 – Processo de Abertura de Conta Comum.

Como podemos verificar, primeiramente o cliente que deseja ter uma conta no banco apresenta um pedido de abertura de conta a um funcionário. Este consultará o cadastro de clientes, o que acarretará o disparo do método `conCpf` em um objeto da classe `Pessoa_Fisica`, passando como parâmetro o CPF do cliente, para determinar se o solicitante já se encontra cadastrado. Se o cliente já estiver registrado, a consulta retornará as informações do cliente, caso contrário retornará um valor significando que o cliente ainda não possui cadastro na instituição. Em seguida, conforme demonstra o fragmento combinado do tipo `opt`, o cadastro do cliente poderá ser atualizado, caso necessário, podendo gerar uma nova instância da classe `Cliente`, se o solicitante não tiver cadastro, ou simplesmente atualizar os dados do mesmo, se houver necessidade de alguma modificação. Como o processo de manutenção de cadastro de clientes já se encontra modelado em outro diagrama, este foi referenciado por meio de um uso de interação, o que está de acordo com o diagrama de casos de uso, uma vez que existe uma associação de extensão entre o caso de uso **Abrir Conta Comum** e o caso de uso **Manter Cliente**.

Depois disso, o cliente escolherá uma senha para a nova conta a ser aberta. O funcionário a seguir solicitará o serviço de abertura de conta por meio da interface do sistema, que repassará o pedido ao controlador que irá então disparar o método `abrir_Conta`, gerando um novo objeto da classe `Conta_Comum`. Se a operação for realizada com sucesso o controlador ordenará então a emissão do cartão da conta.

Finalmente, o cliente deve fornecer um valor inicial para depósito, e o funcionário, por sua vez, solicitará a execução do serviço de **Realizar Depósito**, concluindo o processo. Uma vez que o processo de **Realizar Depósito** é um caso de uso independente, apenas referenciamos esse processo por meio de um uso de interação sobreposta sobre as linhas de vida dos objetos do diagrama e representamos a solicitação de sua execução por meio do disparo de uma mensagem pelo funcionário.

Se observarmos o diagrama de caso de uso desse sistema, perceberemos haver uma associação de inclusão entre os casos de uso **Abrir Conta Comum** e **Realizar Depósito**. Por esse motivo, o processo está só referenciado e não é necessário nenhum teste, pois uma associação de inclusão indica uma obrigatoriedade, e por isso só é preciso colocar o uso de interação sobre as linhas de vida dos objetos.

7.16.3 Exemplo de Diagrama de Sequência – Realizar Depósito

Neste processo, os componentes são os mesmos do diagrama anterior, exceto pelo objeto da classe `Pessoa_Fisica`, desnecessário aqui, porque um cliente não precisa se identificar para depositar um valor. A figura 7.26 apresenta o diagrama de sequência referente ao processo de **Realizar Depósito**.

Nesse diagrama, o cliente informa ao funcionário (poderia ser um caixa eletrônico também, e nesse caso não haveria esse segundo ator) o número da conta que deverá receber o valor a ser depositado. O funcionário em resposta verificará se essa conta existe, solicitando sua consulta à interface do sistema, esta repassará o número da conta ao controlador que, em resposta, disparará o método `consultar_Conta`. Se a conta informada for encontrada, o funcionário solicitará o valor a ser depositado e o cliente o fornecerá. O funcionário então informará à interface do sistema a quantidade do valor a depositar. Esse valor será repassado ao controlador, que disparará o método `depositar_valor`. Se esse método for realizado com sucesso, o controlador pedirá à interface que apresente uma mensagem informando que a operação foi concluída.

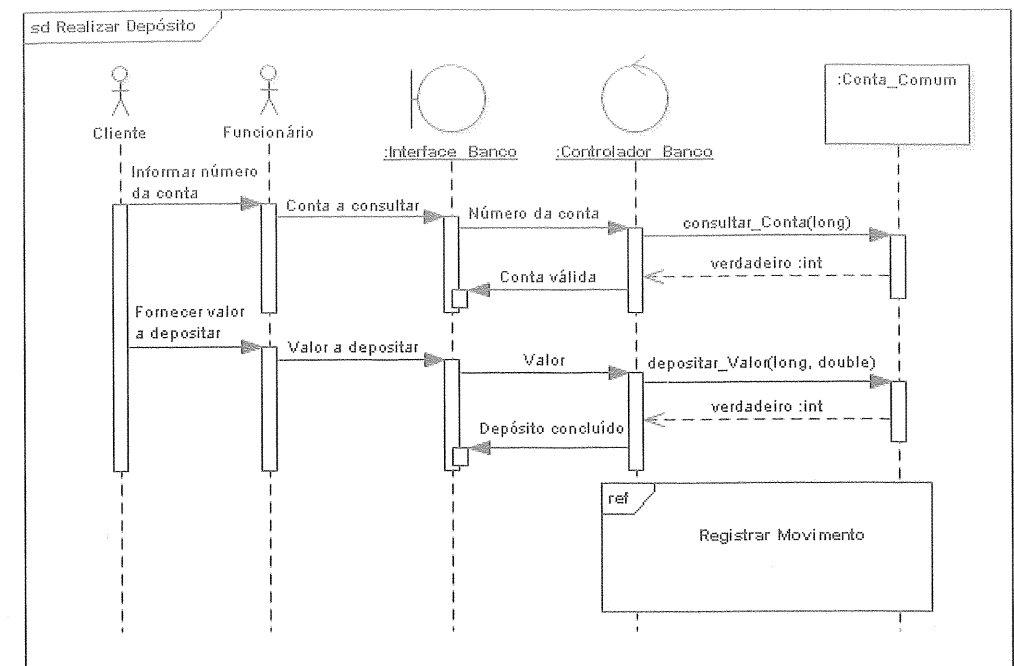


Figura 7.26 – Processo de Depósito.

O sistema deverá ainda registrar o movimento realizado sobre a conta em questão, e para isso ele faz referência ao processo de **Registrar Movimento**, que, como foi definido no diagrama de casos de uso, tem uma relação de inclusão com o caso de uso **Realizar Depósito**. Assim, é necessário apenas posicioná-lo sobre as linhas de vida dos objetos e, como esse é um processo interno do sistema, não é necessário o disparo de nenhuma mensagem por parte dos atores para que ele seja executado.

7.16.4 Exemplo de Diagrama de Sequência – Emitir Extrato

Este exemplo envolve o ator `Cliente` e instâncias das classes `Interface_Banco`, `Controlador_Banco`, `Conta_Comum` e `Movimento`. Não é necessária a interação com o funcionário, uma vez que a emissão do extrato pode ser feita diretamente por um caixa eletrônico. A figura 7.27 apresenta o diagrama de sequência referente ao processo de emissão de extrato.

Nesse processo, o cliente fornece o número de seu cartão à interface. Esta repassa o número informado ao controlador, que ordenará o disparo do método `consultar_Conta` em um objeto da classe `Conta_Comum` para determinar se existe uma conta corrente comum com um número de conta igual ao informado. Em caso positivo, o método retornará um valor indicando que a conta é válida para o objeto de controle.

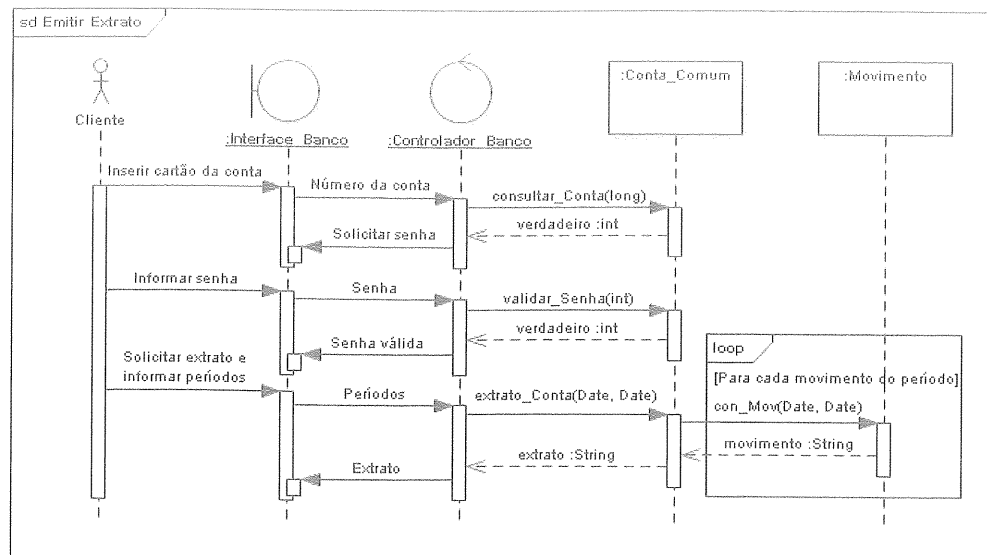


Figura 7.27 – Processo de Emissão de Extrato.

No caso de a conta ser válida, o objeto da classe controladora informará ao objeto da classe de fronteira para solicitar a senha da conta ao cliente. O cliente, por sua vez, digitará sua senha e a interface a enviará para o objeto da classe controladora, que, por sua vez, chamará o método de validação de senha, `validar_Senha` no objeto da classe `Conta_Comum`, passando como parâmetro a senha informada. O método de validação retornará um valor indicando se a senha é válida ou não.

Caso a senha esteja correta, o cliente selecionará a opção de extrato e fornecerá os períodos que determinarão o intervalo desejado do relatório. Ao receber essas informações, a interface as retransmitirá ao objeto de controle, que irá então disparar o método `extrato_Conta` no objeto da classe `Conta_Comum` e esta por sua vez disparará o método `con_Mov` em cada objeto da classe `Movimento` que esteja associado à conta informada e esteja dentro do período solicitado, conforme demonstra o fragmento combinado do tipo `loop` e a condição de guarda a ele associada, retornando uma `String` com os dados de cada movimento.

Os valores contidos nas instâncias que satisfizerem à condição imposta pelo cliente serão então retornados à interface com a instrução fornecida pelo controlador de serem impressas e entregues ao cliente.

7.17 Exercícios Propostos

Aqui será dada continuidade aos exercícios propostos anteriormente, retomando a modelagem dos sistemas iniciados nos capítulos anteriores. Abordaremos em cada exercício apenas um dos processos considerados mais importante em cada sistema.

7.17.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Desenvolva o diagrama de sequência para o processo de venda de ingressos de cinema, de acordo com os seguintes requisitos:

- Ao selecionar a opção de venda de ingressos, o sistema carrega todas as sessões ainda não encerradas, detalhando horário, o filme apresentado e o número da sala.
- O cliente escolherá entre as opções a sessão que deseja assistir e o funcionário irá gerar o ingresso referente a mesma.

7.17.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

Desenvolva o diagrama de sequência referente ao processo de pagamento de mensalidade do sistema de controle de clube social, de acordo com os seguintes requisitos:

- O sócio deve informar o número de seu cartão de sócio ao atendente, que consultará no sistema a mensalidade do mês a ser pago e, se existirem, as possíveis mensalidades em atraso, já com o valor acrescido de juros até o dia da consulta.
- O sócio então poderá escolher quais mensalidades pagar, caso haja mais de uma. O clube exige que sejam pagas primeiro as mensalidades com maior atraso.
- Ao ser realizado o pagamento, o atendente quitará as mensalidades em questão.

7.17.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Desenvolva o Diagrama de Sequência para o processo de locação de veículo do sistema de controle de aluguel de carros, levando em consideração os seguintes requisitos:

- Primeiramente, o funcionário deve selecionar o cliente que está locando o automóvel em uma lista. Para isso, ao selecionar a opção locação, o sistema deve carregar todos os clientes da empresa.

- Em seguida, o funcionário deve informar a placa do veículo que o cliente deseja locar, selecionando o automóvel também em uma lista, que também foi carregada pelo sistema quando o processo foi iniciado.
- Ao selecionar a placa, o sistema apresentará detalhes do automóvel, como ano, cor e quilometragem, além do modelo e marca do veículo, que deverão ser consultados pelo sistema.
- Finalmente, caso o cliente queira realmente locar o veículo selecionado, ele informará o período em que locará o veículo, para qual finalidade e para onde irá. Isso irá gerar uma fatura de locação, que o cliente deverá pagar para concluir a locação.

7.17.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

Desenvolva o diagrama de sequência para o processo de Realizar Leilão do sistema de leilão via internet de acordo com os seguintes requisitos:

- Ao selecionar a opção realizar leilão, o leiloeiro faz com que o sistema selecione todos os leilões não encerrados, apresentando-os ao leiloeiro.
- O leiloeiro deve então escolher o leilão que deseja realizar, o que faz com que o sistema carregue todos os itens a serem leiloados.
- Enquanto houver itens a ser leiloados, o leiloeiro deve buscar o próximo item a ser leiloadado e apresentá-lo na página do leilão, destacando a foto do item, uma breve descrição do produto e o lance mínimo para arrematá-lo.
- Em seguida, o leiloeiro deve aguardar um determinado tempo. Os lances serão ofertados pelos participantes.
- Cada lance que suplante o anterior deve ser anunciado, destacando o participante que o fez. Sempre que um lance suplantar o anterior deve ser registrado.
- Quando os lances se esgotarem, depois de um tempo de espera após a oferta do último lance, o vencedor é anunciado, e o item, marcado como arrematado.
- Ao concluírem-se os itens, o leiloeiro deve finalizar o leilão.

7.17.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

Desenvolva o diagrama de sequência referente ao processo de pagamento das diárias para o sistema de hotelaria sabendo que:

- O hóspede se dirige ao funcionário e informa os quartos que deseja pagar.
- O funcionário, por meio do sistema, deve então buscar todas as diárias ainda não pagas relativas ao quarto e apresentar ao hóspede.
- Ao realizar o pagamento, as diárias serão quitadas, podendo o hóspede permanecer no hotel ou encerrar sua estadia.
- Caso o hóspede tenha solicitado algum serviço ou consumido algum item de frigobar, deverá pagá-los igualmente.

7.18 Solução dos Exercícios

A seguir apresentaremos as soluções relativas a cada exercício proposto na seção anterior. Sugerimos que o leitor acompanhe a explicação de cada exercício consultando o diagrama de casos de uso e o diagrama de classes de cada um dos sistemas para ter uma compreensão mais completa da solução.

7.18.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Podemos visualizar o detalhamento desse processo na figura 7.28. Nesse processo participam o ator Funcionário e instâncias da classe de fronteira e de controle, além de objetos das classes de entidade Sessão, Sala, Filme e Ingresso. O processo será explanado a seguir.

Nesse processo, o funcionário seleciona a opção de venda de ingresso na interface do sistema. Esse evento é repassado à controladora, que iniciará um laço, representado pelo fragmento combinado do tipo loop, no qual, para cada sessão ainda não encerrada, será disparado o método selSessao. Esse método consulta a sala associada à sessão por meio do método conSala e, em seguida, o filme nela apresentado, por meio do método conFilme. Essas informações serão então retornadas à controladora, que ordenará sua apresentação na interface.

Com base nessas informações, o funcionário informará a sessão desejada e confirmará. Isso fará com que a interface informe à controladora a sessão escolhida, e esta chamará o método gerIng para gerar um novo objeto da classe Ingresso. Se o método for concluído com sucesso, a controladora pedirá à interface para emitir o ingresso que deverá ser entregue ao cliente pelo funcionário.

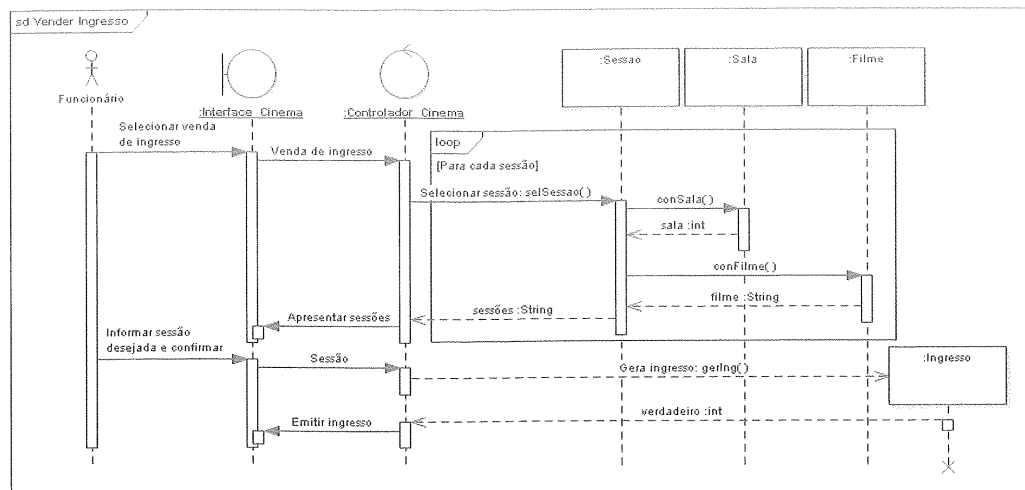


Figura 7.28 – Processo de Venda de Ingresso.

7.18.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

Esse processo está detalhado na figura 7.29. Aqui, participam os atores Sócio e Clube, que representa os funcionários do clube. No processo interagem também as instâncias da classe de fronteira e de controle, além de objetos das classes de entidade Socio e Mensalidade. O processo será detalhado a seguir.

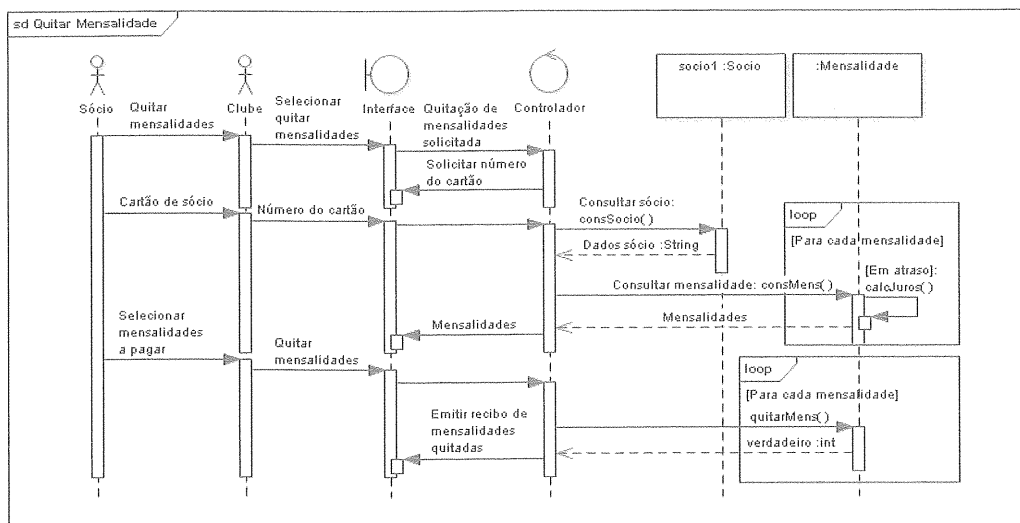


Figura 7.29 – Processo de Pagamento de Mensalidade.

Nesse processo, o sócio se encaminha a um funcionário e o informa que deseja quitar mensalidades. O funcionário então solicita esse serviço à interface, que repassa o pedido à controladora, que, em resposta, pedirá que a interface apresente um formulário no qual se deve inserir o número do cartão do sócio.

O sócio então fornecerá o número de seu cartão, que será repassado ao controlador. Em posse desse número, o controlador chamará o método consSocio para consultar o sócio. Esse método, caso o sócio seja encontrado, retornará os dados do mesmo. Em seguida o controlador consultará as mensalidades do sócio por meio do disparo do método consMens em cada objeto associado ao sócio consultado, como demonstra o fragmento combinado do tipo loop. Dentro desse laço existe ainda um teste, determinado por uma condição de guarda (texto entre colchetes), que determina que, se a mensalidade estiver em atraso, deve-se disparar o método calcJuros. Observe que esta é uma autochamada. O conteúdo de cada mensalidade será então retornado à controladora, que as enviará para serem apresentadas na interface.

Com base nas informações apresentadas, o sócio escolherá quais mensalidades deseja pagar e o funcionário mandará quitá-las através da interface, que as repassará ao controlador. Este então disparará o método quitarMens para cada uma das mensalidades selecionadas, conforme demonstra o fragmento combinado do tipo loop, definindo-as como quitadas. Depois de receber o retorno de que as mensalidades foram quitadas, o controlador mandará a interface emitir o recibo de quitação.

7.18.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Podemos visualizar a solução para esse processo na figura 7.30. No processo participa somente o ator Funcionário (poderíamos ter colocado também o ator Cliente, mas o diagrama se tornaria largo demais). Neste processo interagem também as instâncias da classe de fronteira e de controle, além de objetos das classes de entidade Cliente, Automovel, Modelo, Marca e Locacao. A seguir explicaremos o processo.

O processo se inicia quando o funcionário seleciona a opção de locar veículo na interface. Esse evento é repassado por ela à controladora, que disparará o método conCli para consultar cada cliente registrado no sistema, conforme demonstra o fragmento combinado do tipo loop. Após a execução desse laço, a controladora terá recebido os dados de todos os clientes registrados.

Em seguida, a controladora inicia outro laço para selecionar todos os automóveis disponíveis para locação, disparando o método conAuto para cada automóvel em condições de locação, conforme demonstra o fragmento combinado do tipo

loop. Esse método consulta o modelo do automóvel consultado, por meio do método `conMod` e este, por sua vez, consulta a marca desse modelo por meio do método `conMarca`. Esses dados são então retornados à controladora, que os manda apresentar na interface juntamente com o formulário de locação de veículos.

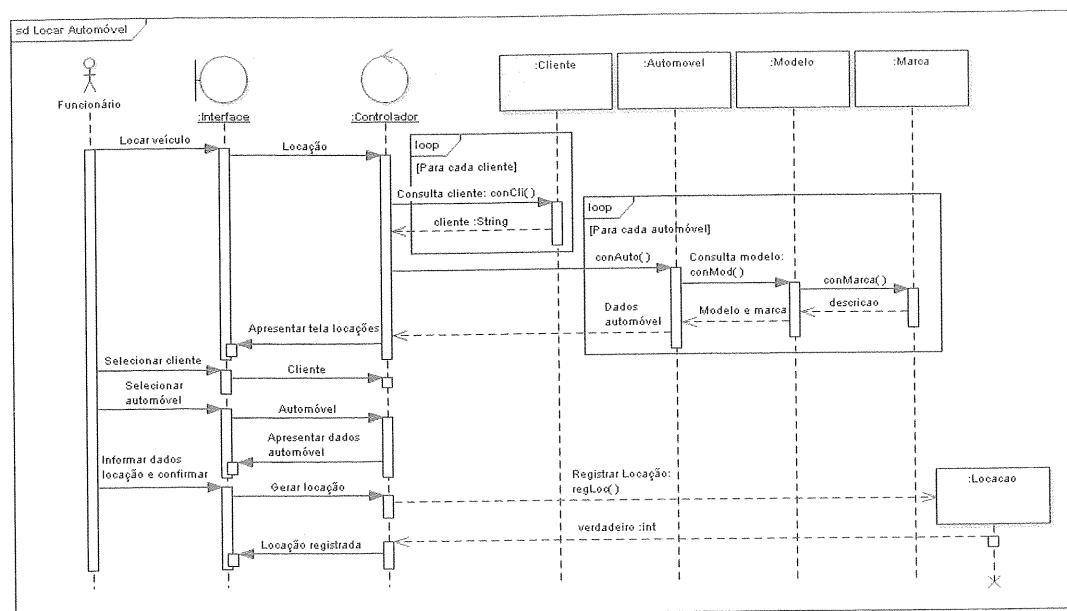


Figura 7.30 – Processo de Locação de Veículo.

O funcionário seleciona o cliente que deseja locar o veículo e em seguida, o veículo desejado. Isso é feito diretamente na interface. A seguir, o funcionário deve informar os dados da locação, como o período de locação, para qual finalidade e para onde o cliente pretende ir. A confirmação da locação na interface fará com que esta avise a controladora da ocorrência desse evento, o que fará com que a controladora dispare o método `regLoc` para registrar a nova locação, instanciando um novo objeto da classe `Locacao` e, caso o método seja executado com sucesso, a controladora mandará a interface informar que a locação foi realizada e que o veículo está liberado.

7.18.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

Na figura 7.31 podemos visualizar o processo de **Realizar Leilão**. O processo conta com a participação dos atores **Leiloeiro** e **Participante**, que representa todos os possíveis participantes do leilão. Além disso, o processo envolve instâncias da classe de fronteira e de controle, bem como objetos das classes de entidade `Leilao`, `Item_Leilao` e `Lance`. A seguir detalharemos os passos desse processo.

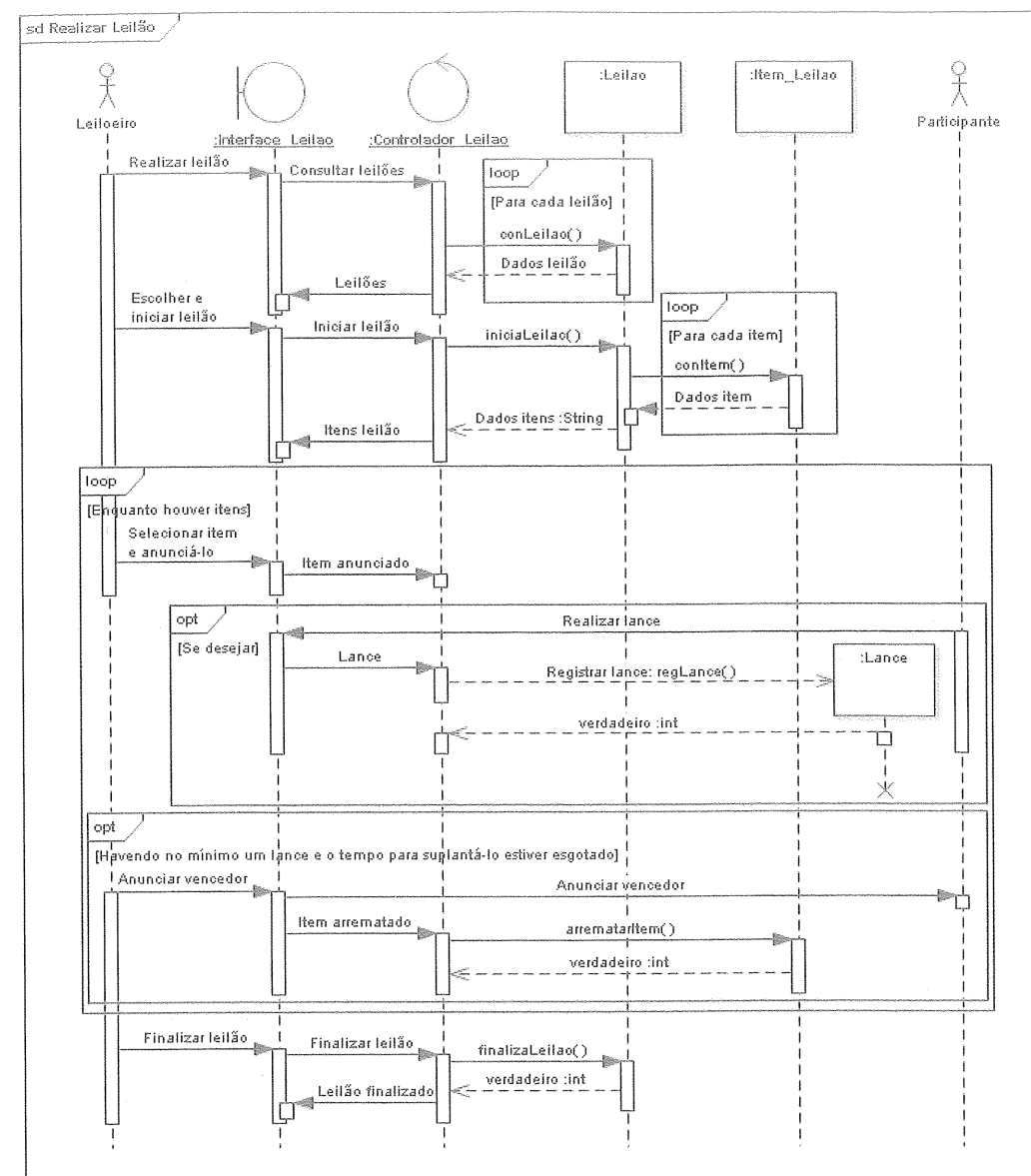


Figura 7.31 – Processo de Realizar Leilão.

O leiloeiro inicia o processo selecionando a opção **Realizar Leilão** na interface do sistema, o que faz com que esta avise a controladora de que é necessário apresentar todos os leilões ainda não encerrados para o leiloeiro. A controladora dispara o método `conLeilao` em todos os objetos da classe `Leilao` ainda não encerrados, para retornar seus dados. Observe que, como esse método deve ser disparado em muitos objetos, esse laço foi representado por meio de um fragmento combinado do tipo `loop`.

Ao receber essa listagem, o leiloeiro deve escolher um leilão e iniciá-lo. Essa ação faz com que a interface avise o controlador deste evento, causando o disparo do método `iniciaLeilao` em um objeto da classe `Leilao`. Esse método precisa selecionar todos os itens da classe `Item_Leilao` associados ao objeto escolhido da classe `Leilao`. Assim, ele dispara o método `conItem` em cada objeto da classe `Item_Leilao` associado ao objeto `Leilao` selecionado, como demonstra o fragmento combinado do tipo `loop`. Os itens do leilão iniciado são então apresentados pela interface ao leiloeiro.

A partir desse momento o processo entra em um laço que será somente concluído quando não houver mais itens a leiloar, como podemos perceber ao observarmos o fragmento combinado do tipo loop, que envolve todos os componentes do processo, contendo a condição de guarda “Enquanto houver itens”:

Assim, enquanto houver itens, o leiloeiro selecionará um deles de acordo com a ordem e irá anunciá-lo por meio da interface. Como se trata de um leilão via internet, todos os participantes serão notificados de qual item está sendo anunciado e poderão, se assim o desejarem, oferecer lances.

O fragmento combinado do tipo `opt`, logo a seguir, modela a situação em que o participante deseja oferecer um lance para um item. O lance oferecido deve ser repassado pela interface à controladora, que disparará o método `reglance` para registrá-lo, criando um novo objeto da classe `Lance`.

Já o segundo fragmento combinado do tipo `opt`, modela a situação em que, tendo ocorrido ao menos um lance e o tempo para suplantá-lo estiver esgotado, o leiloeiro deverá anunciar o participante vencedor, através da interface, que se encarregará de comunicar o participante de sua vitória, bem como anunciar isto na página. Ao mesmo tempo ela notificará isto à controladora e esta se encarregará de definir o item como arrematado, por meio do método `arrematarItem`.

Finalmente, quando todos os itens tiverem sido ofertados, o leiloeiro solicitará o encerramento do leilão à interface, que repassará o pedido à controladora, causando o disparo do método `finalizaLeilao`, que definirá o leilão como encerrado.

7.18.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

A figura 7.32 apresenta a solução desse problema. Além do ator funcionário, o diagrama conta com instâncias da classe de fronteira e de controle, bem como objetos das classes de entidade Quarto, Aluga, Hospede e Diaria. A seguir detalharemos os passos desse processo.

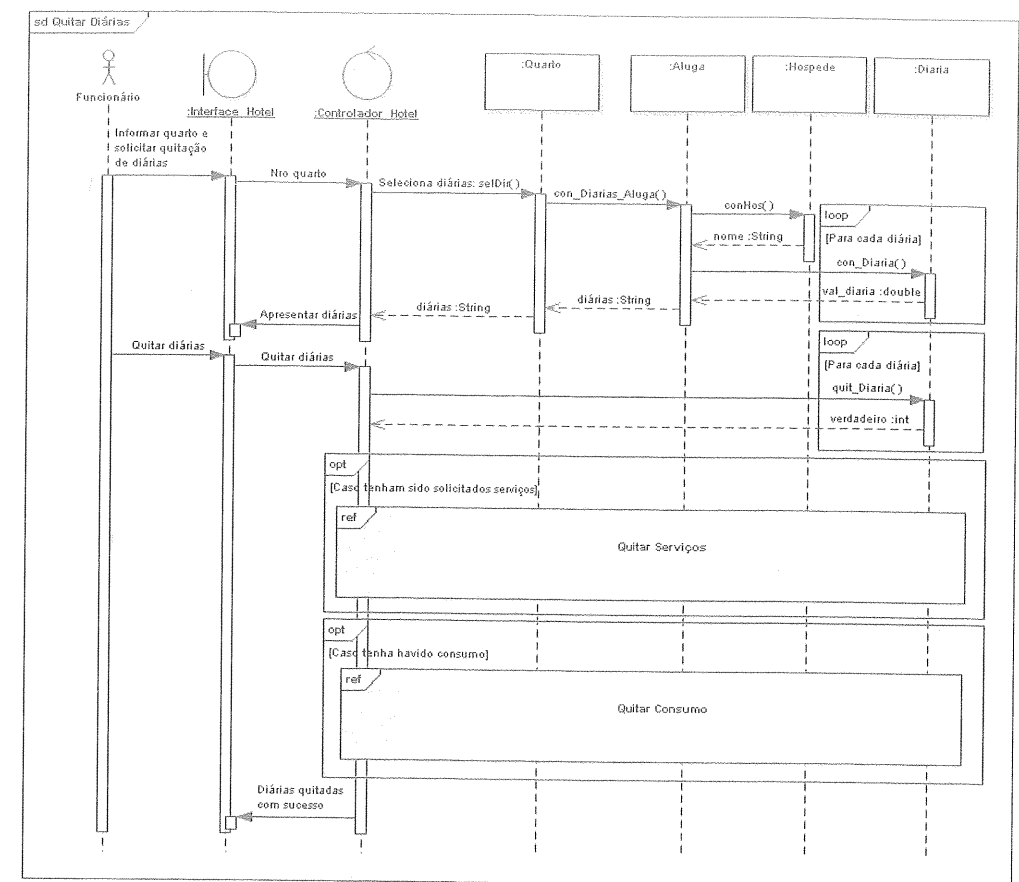


Figura 7.32 – Processo de Pagamento de Diárias.

Nesse processo, o funcionário informa o número do quarto e solicita o serviço de quitação de diárias à interface. Esta repassa o número do quarto para a controladora que, em resposta, dispara o método `seLDi` em um objeto da classe `Quarto` para selecionar as diárias devidas pelo hóspede.

O método `seldir` deve retornar o nome do hóspede que alugou o quarto (que não significa que seja o mesmo que o está ocupando) junto com as diárias devidas. Para isso é necessário buscar informações contidas nos objetos `Aluga`, `Hospede` e `Diaria`. Assim, esse método primeiramente dispara o método `con_Diarias_Aluga` em um objeto da classe `Aluga`, para que este, por sua vez, consulte o hóspede ao qual está associado, por meio do método `conHos` e, depois de obter seu retorno na forma de uma `String` contendo o nome do hóspede, dispare o método `con_Diaria` em cada objeto da classe `Diaria` a ele associado, como demonstra o fragmento combinado do tipo `loop`, retornando o valor da diária. De posse desses valores, o método `con_Diarias_Aluga` os retornará na forma de uma `String` para o método `seldir` que o chamou, que por sua vez os retornará ao controlador.

Ao obter esse relatório, o funcionário selecionará a opção de quitar diárias na interface, que repassará esse evento ao controlador, que, em resposta, disparará o método `quit_Diaria` em cada objeto associado ao quarto alugado, como demonstra o fragmento combinado do tipo `loop`. Esse método retornará verdadeiro para cada execução bem-sucedida.

Caso tenham sido solicitados serviços ou tenha havido consumo do frigobar, o hóspede deverá quitá-los também. Como os processos de **Quitar Serviços** e **Quitar Consumo** estão modelados em diagramas separados, eles foram apenas referenciados por meio de um uso de interação. Observe que a execução desses usos de interação depende de uma condição estabelecida por um fragmento combinado do tipo `opt` para cada uma delas, o que significa que poderão ser executadas ou não.

Finalmente, depois de quitar as diárias e eventualmente os possíveis serviços solicitados e/ou consumos ocorridos, o sistema apresentará a mensagem de que as diárias foram quitadas.



CAPÍTULO 8

Diagrama de Comunicação

O diagrama de comunicação era conhecido como diagrama de colaboração até a versão 1.5 da UML, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0. O diagrama está amplamente associado ao diagrama de sequência: na verdade, um complementa o outro. As informações mostradas no diagrama de comunicação são, com frequência, praticamente as mesmas apresentadas no diagrama de sequência, porém com um enfoque diferente, visto que esse diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante um processo.

Por ser muito semelhante ao diagrama de sequência, o diagrama de comunicação utiliza muitos de seus componentes, como atores e objetos, incluindo seus estereótipos de fronteira e controle. No entanto os objetos no diagrama de comunicação não têm linhas de vida. Além disso, esse diagrama não suporta ocorrências de interação ou fragmentos combinados como o diagrama de sequência, por isso é utilizado para a modelagem de processos mais simples.

Da mesma forma que no diagrama de sequência, um diagrama de comunicação enfoca um processo, normalmente baseado em um caso de uso. As semelhanças entre ambos são tão grandes que existem até mesmo ferramentas CASE capazes de gerar um dos diagramas a partir do outro. Nas seções seguintes detalharemos o diagrama de comunicação.

8.1 Lifelines

Os lifelines do diagrama de comunicação representam o mesmo que no diagrama de sequência, ou seja, participantes individuais de uma interação e, em geral, representam instâncias de classes que participam do processo. No entanto, diferentemente do diagrama de sequência, e apesar do nome, os lifelines utilizados por esse diagrama não têm linha de vida ou foco de controle, tendo a

mesma representação utilizada no diagrama de objetos e as mesmas regras de nomenclatura, mas sem haver um detalhamento dos valores de seus atributos. Uma vez que, na prática, um lifeline representa em geral um objeto, optaremos por usar essa nomenclatura. A figura 8.1 apresenta um exemplo de lifeline.



Figura 8.1 – Exemplo de Lifeline.

Como podemos notar, da mesma forma que no diagrama de sequência, o texto do retângulo (lifeline) informa o nome do objeto e o da classe ao qual ele pertence, nessa ordem. As duas informações vêm separadas por um símbolo de dois pontos. Assim, o objeto representado na figura chama-se pesfis1, e é uma instância da classe Pessoa_Fisica.

8.2 Vínculos

Um vínculo nada mais é do que uma instância de uma associação definida no diagrama de classes e identifica uma ligação entre dois objetos envolvidos em um processo. Assim, a existência de um vínculo é caracterizada sempre que dois objetos colaboram entre si dentro de um processo, seja pelo envio ou recebimento de mensagens, ou ambos. Tal vínculo é representado por uma linha unindo os dois objetos. A figura 8.2 apresenta um exemplo de vínculo entre dois objetos, demonstrando que um objeto da classe de controle Controlador_Banco está vinculado a um objeto da classe de entidade Pessoa_Fisica. Observe que os mesmos estereótipos aplicados no diagrama de sequência podem ser igualmente aplicados aos objetos no diagrama de comunicação.



Figura 8.2 – Exemplo de Vínculo entre Objetos.

8.3 Mensagens

As mensagens identificadas nesse diagrama são as mesmas definidas no de sequência, e em geral representam chamadas de métodos. No entanto, não existe uma preocupação com a temporalidade, ou seja, a ordem em que elas são chamadas não é relevante, o que importa é que são disparadas entre os elementos envolvidos no processo. A única noção temporal passada por esse diagrama é a numeração das mensagens, indicando a ordem em que ocorrem. Uma mensagem é representada por uma seta indicativa da direção para onde a mensagem foi enviada, ou seja, a ponta da seta indica o objeto que receberá a mensagem, enquanto o objeto na direção oposta será o responsável por seu envio.

É necessário primeiro existir um vínculo entre os objetos para que as mensagens possam ser inseridas. Um único vínculo pode suportar muitas mensagens, podendo estas ser mensagens de retorno, e não é possível existir mais de um vínculo entre os mesmos objetos. A figura 8.3 mostra um exemplo de mensagens trocadas entre objetos vinculados.

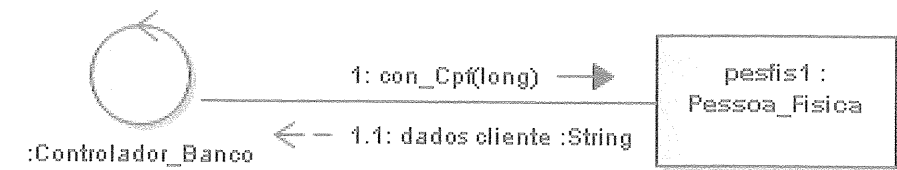


Figura 8.3 – Exemplo de mensagens.

Esse exemplo demonstra que o objeto da classe controladora dispara uma mensagem para que seja executado o método conCpf, passando como parâmetro o CPF a ser consultado no objeto pesfis1 da classe Pessoa_Fisica e este, em resposta, envia uma mensagem contendo os dados do cliente consultado.

8.4 Autochamada

Um objeto pode disparar uma mensagem em si próprio, o que é conhecido como autochamada, onde a mensagem parte do objeto e retorna ao próprio objeto. A figura 8.4 apresenta um exemplo de autochamada em um objeto.

Na figura 8.4, o objeto pesfis1 envia uma mensagem para si mesmo, solicitando o disparo do método de validação de CPF.

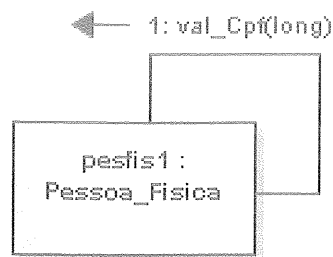


Figura 8.4 – Autochamada.

8.5 Atores

Os atores apresentados nesse diagrama são exatamente os mesmos utilizados no diagrama de sequência no diagrama de casos de uso, representando as entidades externas que interagem com o sistema de alguma forma. Os atores desse diagrama não têm linha de vida ou foco de controle como no diagrama de sequência, sendo representados exatamente da mesma forma que no diagrama de casos de uso.

Um ator também tem vínculos com outros objetos ou atores e envia e recebe mensagens por meio desses vínculos, da mesma forma que os demais objetos. A figura 8.5 apresenta um exemplo de um ator interagindo com outros objetos.

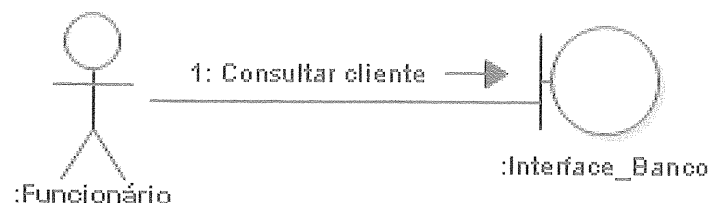


Figura 8.5 – Ator Interagindo com Objeto.

Aqui demonstramos que existe um vínculo entre o ator Funcionário e um objeto da classe Interface_Banco. Por meio desse vínculo o ator envia uma mensagem à interface solicitando a consulta de um cliente. Essa mensagem não representa um disparo de método, apenas a ocorrência de um evento sobre a interface causado pelo ator.

8.6 Exemplo de diagrama de comunicação – Processo de Emissão de Saldo

Nesta seção apresentaremos um exemplo de diagrama de comunicação, referindo-se este ao processo de emissão de saldo, representado pelo caso de uso Emitir Saldo e já modelado no diagrama de sequência. A figura 8.6 apresenta o diagrama referente a esse processo.

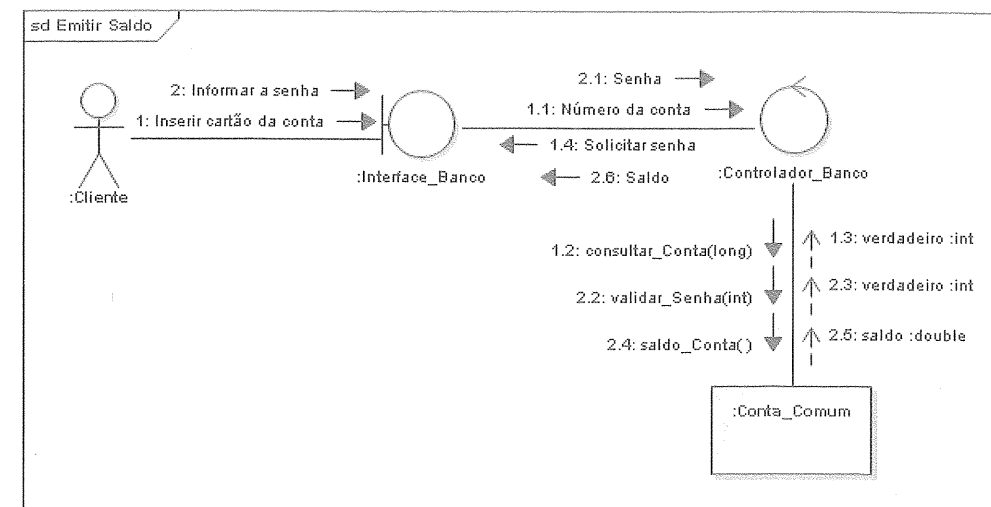


Figura 8.6 – Processo de Emissão de Saldo.

Nessa figura interagem o ator Cliente e objetos das classes Interface_Banco, Controlador_Banco e Conta_Comum. O ator tem um vínculo com a interface, esta com o objeto da classe controladora e essa última com o objeto da classe Conta_Comum.

Existem aqui dois grupos de mensagens, conforme podemos observar pela numeração das mesmas. O primeiro grupo inicia-se com o ator fornecendo um número de conta à interface. Esta, por sua vez, repassará esse número à controladora e esta, por meio do vínculo com o objeto da classe Conta_Comum, disparará por meio de uma mensagem o método consultar_Conta, que recebe como parâmetro o número da conta. Em resposta, o objeto da classe Conta_Comum retornará verdadeiro através de uma mensagem de retorno, se a conta existir, ou falso, em caso contrário.

O segundo grupo também é iniciado pelo ator, que dessa vez fornece a senha da conta por meio do vínculo com a interface. A interface repassa essa senha para a controladora, e esta dispara o método validar_Senha no objeto da classe Conta_Comum, passando como parâmetro a senha recebida. Em resposta, o objeto da classe Conta_Comum retornará verdadeiro, se a senha for válida, ou falso, em caso

contrário. Em seguida, o controlador solicitará a execução do método `saldo_Conta` no objeto da classe `Conta_Comum` e este, em resposta, enviará uma mensagem de retorno contendo o valor armazenado na conta consultada. O controlador toma esse valor e solicita sua apresentação na interface, encerrando o processo.

8.7 Condições de Guarda e Iterações

Condições de Guarda são textos entre colchetes que estabelecem condições ou validações para que uma mensagem possa ser enviada. Já iterações representam uma situação em que a mensagem pode ser enviada várias vezes, correspondendo muitas vezes a um laço. As iterações são representadas por um asterisco (*) na frente da mensagem e em geral vêm apoiadas por condições de guarda. Uma vez que o diagrama de comunicação não suporta fragmentos combinados, muitas vezes é necessário lançar mão desse artifício para representar situações opcionais ou laços. A figura 8.7 apresenta um exemplo de mensagem com iteração e condição de guarda enfocando o processo de emissão de extrato.

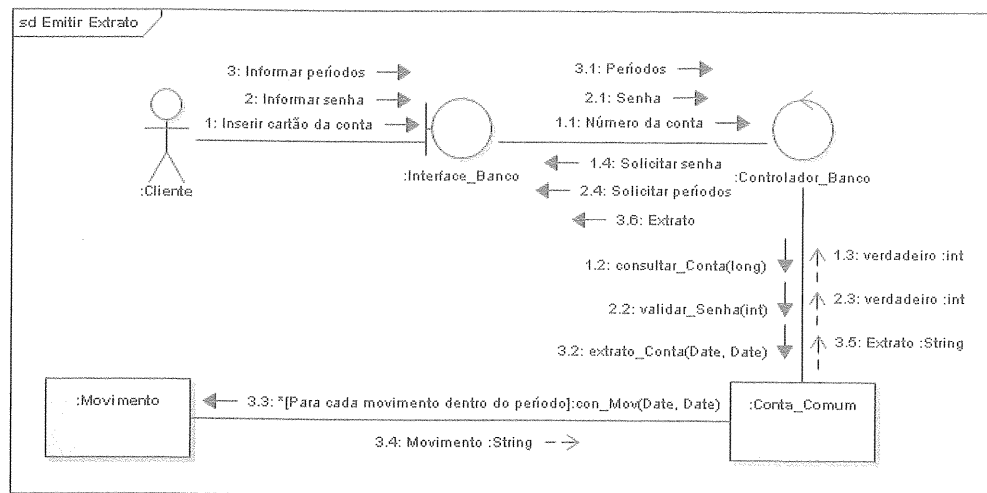


Figura 8.7 – Condição e Iteração no Diagrama de Comunicação – Processo de Emissão de Extrato.

O processo aqui representado refere-se ao caso de uso Emitir Extrato, igualmente já descrito no diagrama de sequência. Esse processo envolve o ator `Cliente` e objetos das classes `Interface_Banco`, `Controlador_Banco`, `Conta_Comum` e `Movimento`.

Esse diagrama representa três grupos de mensagens. O primeiro grupo inicia-se com o cliente inserindo o número do cartão na interface. A interface repassa o número da conta para o controlador e este chama o método `consultar_Conta` passando como parâmetro o número da conta em um objeto da classe `Conta_Comum`.

Em resposta esse objeto enviará uma mensagem de retorno à controladora determinando se a conta foi encontrada (verdadeiro) ou não (falso). Supondo que a conta tenha sido encontrada, o controlador enviará uma mensagem à interface solicitando que esta peça ao cliente a senha da conta.

Quando o cliente fornece a senha à interface inicia-se o segundo grupo de mensagens. A interface repassa a senha para a controladora e esta solicita a validação da senha, por meio do disparo do método `validar_Senha`, no objeto da classe `Conta_Comum`, passando como parâmetro a senha. O objeto da classe `Conta_Comum` retornará uma mensagem contendo verdadeiro se a senha estiver correta. Nesse caso, o controlador pedirá à interface para solicitar os períodos do extrato.

Ao informar os períodos na interface, o cliente inicia o terceiro e último grupo de mensagens. Os períodos são transmitidos à controladora, que disparará o método `extrato_Conta` passando como parâmetro as datas iniciais e finais desejadas pelo cliente em um objeto da classe `Conta_Comum`. O objeto da classe `Conta_Comum` disparará, por sua vez, o método `con_Mov` passando como parâmetro as mesmas datas, em cada objeto da classe `Movimento` que estiver associado ao objeto da classe `Conta_Comum` e estiver dentro do período solicitado. Observe que essa última mensagem se caracteriza por ser uma iteração, como demonstra o asterisco (*) no início da mensagem, ou seja, ela poderá ser disparada muitas vezes, além disso a mensagem é acompanhada pela condição de guarda com o texto “Para cada movimento dentro do período”, que estabelece a condição para que a iteração seja executada. Cada movimento selecionado será retornado ao objeto da classe `Conta_Comum`, que por sua vez os repassará à controladora que os mandará apresentar na interface.

8.8 Exercícios Propostos

Os exercícios aqui propostos são exatamente os mesmos descritos no final do capítulo sobre o diagrama de sequência, uma vez que o objetivo dos dois diagramas é o mesmo. Assim, não achamos necessário inserir a descrição dos exercícios novamente. Passaremos assim diretamente para a solução dos exercícios, excetuando-se o processo de quitar diárias do sistema de controle de hotel, devido a este utilizar ocorrências de interação que não são suportadas pelo diagrama de comunicação, utilizado para modelar processos mais simples. Alternativamente, se o leitor desejar, seria possível construir um diagrama de comunicação abrangendo os três processos de quitar diária, serviço e consumo, ou criar três diagramas separados, um para cada processo.

8.9 Solução dos Exercícios

Na solução dos exercícios inseriremos somente as figuras com a solução destes, uma vez que a descrição das soluções dos processos já se encontram no capítulo sobre o diagrama de sequência. É importante destacar, no entanto, que fragmentos combinados do tipo opt costumam ser representados por condições de guarda nesse diagrama, e fragmentos combinados do tipo loop são representados pelo símbolo de iteração (*).

8.9.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

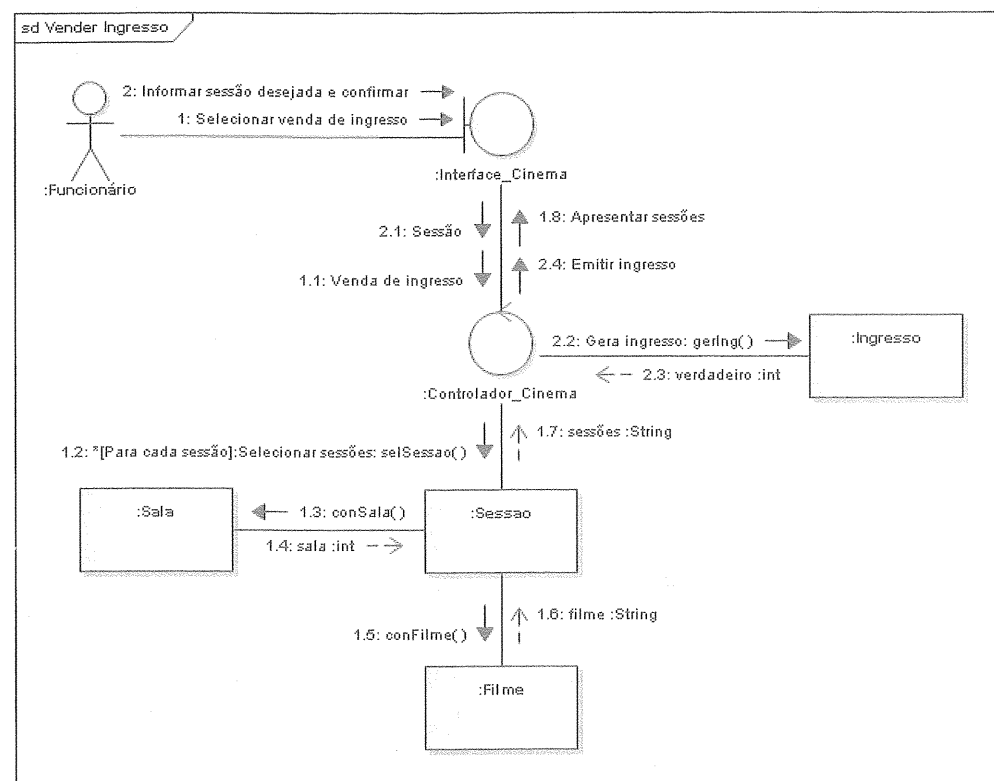


Figura 8.8 – Processo de Venda de Ingressos.

8.9.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

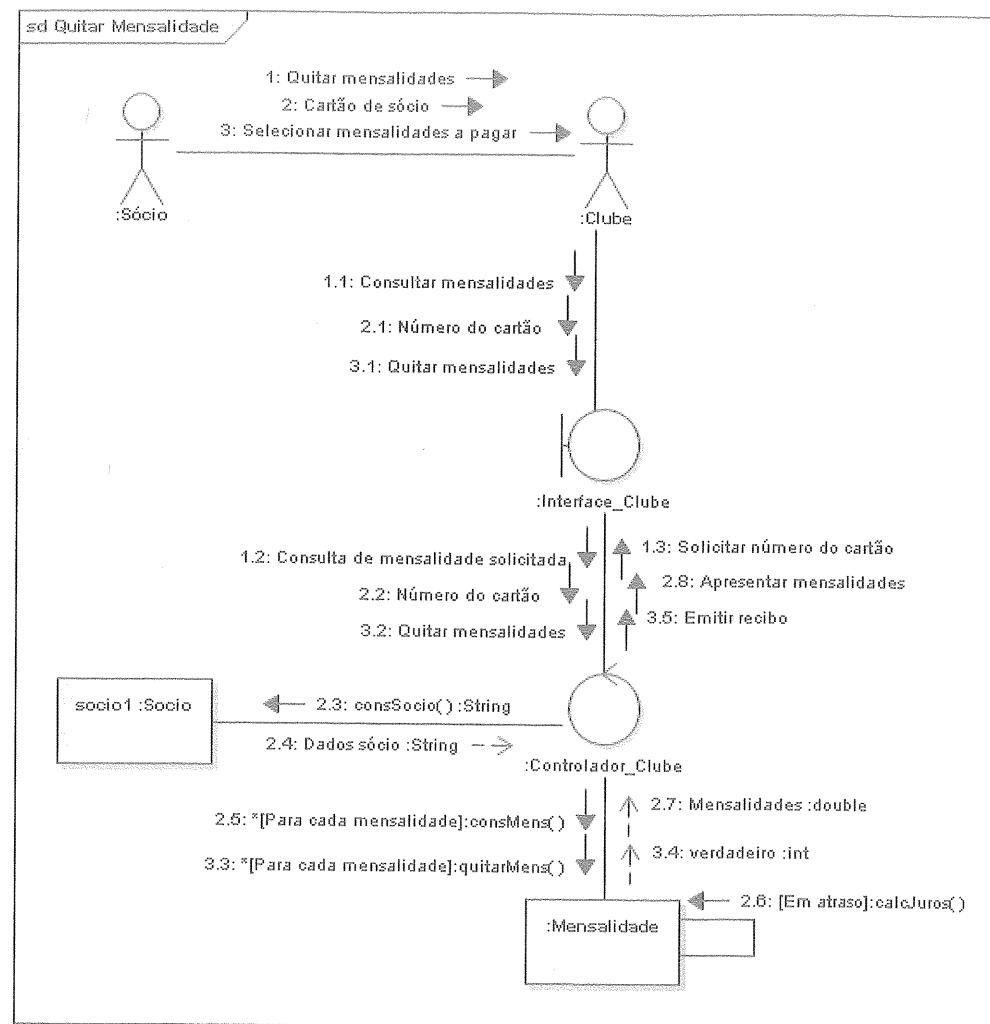


Figura 8.9 – Processo de Pagamento de Mensalidade.

8.9.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

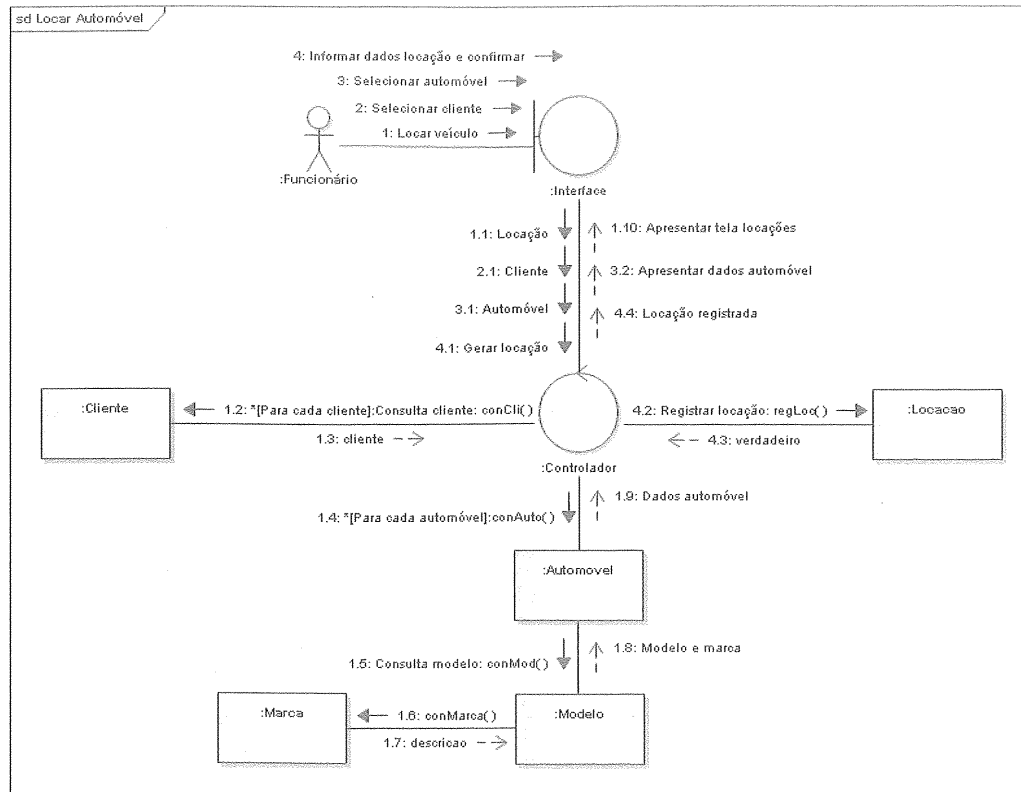


Figura 8.10 – Processo de Locação de Veículo.

8.9.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

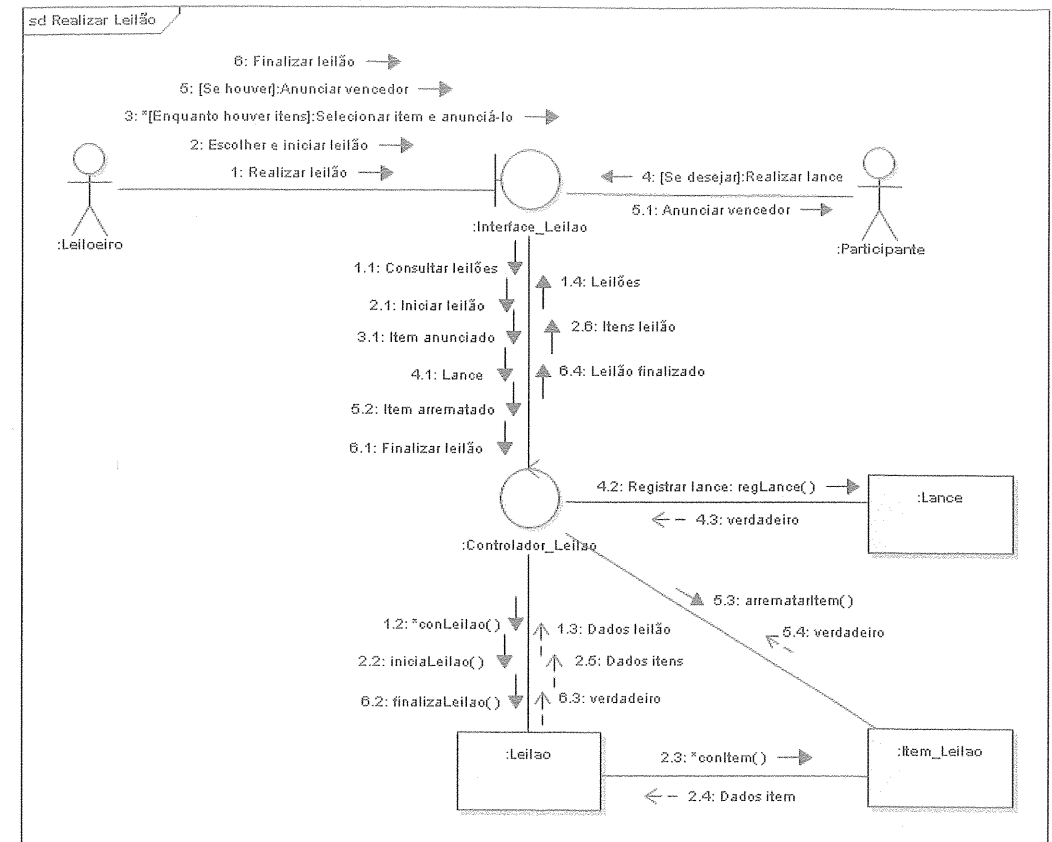


Figura 8.11 – Processo de Realizar Leilão.



CAPÍTULO 9

Diagrama de Máquina de Estados

Esse diagrama era chamado de diagrama de gráfico de estados ou simplesmente diagrama de estados nas versões anteriores da UML, tendo seu nome modificado para diagrama de máquina de estados a partir da versão 2.0. O diagrama de máquina de estados demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado, ou seja, uma máquina de estados. Além de poder ser utilizado para expressar o comportamento de uma parte do sistema, quando é chamado de máquina de estado comportamental também pode ser usado para expressar o protocolo de uso de parte de um sistema, quando identifica uma máquina de estado de protocolo. Neste capítulo nos concentraremos em máquinas de estado comportamentais.

Uma máquina de estados comportamental pode ser usada para especificar o comportamento de vários elementos do modelo. O elemento modelado muitas vezes é uma instância de uma classe. No entanto, pode-se usar esse diagrama para modelar o comportamento de um caso de uso, por exemplo.

9.1 Estado

Um estado representa a situação em que um elemento (muitas vezes um objeto) se encontra em determinado momento durante o período em que participa de um processo. Um objeto pode passar por diversos estados dentro de um mesmo processo. Um estado pode demonstrar:

- A espera pela ocorrência de um evento.
- A reação a um estímulo.
- A execução de alguma atividade.
- A satisfação de alguma condição.

9.1.1 Estado Simples

Um estado simples é um estado que não tem subestados, ou seja, não pode ser subdividido em estados internos. É o tipo mais comum de estado. Por esse motivo, será referido simplesmente como estado ao longo do capítulo. A figura 9.1 apresenta um exemplo de estado simples.

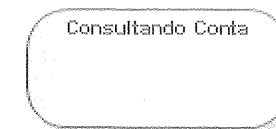


Figura 9.1 – Exemplo de Estado Simples.

O estado ilustrado na figura 9.1 é um dos estados pelos quais passa um objeto da classe `Conta_Comum` (ou pelos objetos de suas subclasses) durante o processo de **Emitir Saldo**, já ilustrado no capítulo sobre o diagrama de sequência. Nesse caso, o objeto da classe `Conta_Comum` encontra-se no estado **Consultando Conta**.

Muitas vezes o estado de um objeto é descrito no gerúndio, como nesse exemplo, uma vez que, em geral, ele está executando uma atividade. No entanto, em alguns casos, o objeto pode estar esperando por um evento, e, portanto, estará em um estado estático ou de espera, não podendo ser descrito no gerúndio. A figura 9.2 apresenta um exemplo de objeto em um estado estático.

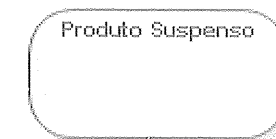


Figura 9.2 – Objeto em Estado Estático.

No exemplo da figura 9.2, um objeto da classe `Produto` foi, por algum motivo, colocado no estado suspenso. Assim, esse objeto está esperando por um evento que o tire do estado atual.

9.2 Transições

Uma transição representa um evento que causa uma mudança no estado de um objeto, gerando um novo estado. Uma transição é representada por uma linha ligando dois estados, contendo uma seta em uma de suas extremidades, apontando para o novo estado gerado. A figura 9.3 apresenta um exemplo de transição entre estados.



Figura 9.3 – Exemplo de Transição entre Estados.

Esse exemplo apresenta uma transição relativa ao processo de Emitir Saldo, enfocando um objeto da classe *Conta_Comum*. Aqui existem dois estados. No primeiro, o objeto da classe *Conta_Comum* está sendo consultado. Depois de a conta ter sido localizada, o cliente deverá informar a senha da conta. Esse evento causa uma transição de estado, em que o objeto da classe *Conta_Comum* passa ao estado *Validando Senha*.

Uma transição pode ou não conter uma descrição (recomenda-se que tenha, para facilitar sua compreensão). A descrição de um evento pode tanto conter uma ordem para realizar alguma tarefa como ser simplesmente uma informação avisando a ocorrência do evento. Além da descrição, uma transição pode também conter condições de guarda e parâmetros.

9.3 Estado Inicial

O estado inicial tem como função somente determinar o início da modelagem dos estados de um elemento. É representado por um círculo preenchido, a partir do qual é gerada uma transição que determina o início do processo. Da mesma forma que nas transições entre os objetos, uma transição de um estado inicial pode conter ou não uma descrição, o que às vezes é útil para identificar o evento que iniciou o processo.

9.4 Estado Final

O estado final tem como função apenas indicar o final dos estados modelados. É representado por um círculo não-preenchido envolvendo um segundo círculo preenchido. A figura 9.4 apresenta um exemplo de estados inicial e final.

Nesse exemplo, usamos o componente nota para identificar os estados iniciais e finais desse diagrama. O diagrama em questão representa os estados pelos quais passa um objeto da classe *Conta_Comum* durante o processo de emissão de saldo. Observe que, a partir do estado inicial, é gerada a transição que dá início aos estados aqui representados, caracterizada pelo fornecimento do número da conta, o que

produz o estado *Consultando Conta*. Depois de a conta ter sido consultada, um novo evento é gerado, caracterizando-se pelo fornecimento da senha da conta, o que produz o estado *Validando Senha*. Quando a validação for concluída, é solicitada a emissão do saldo, o que cria o estado *Consultando Saldo*. No momento em que esse estado é concluído, é gerada uma transição contendo o saldo a ser apresentado. Essa última transição conclui o diagrama, conforme demonstra o estado final.

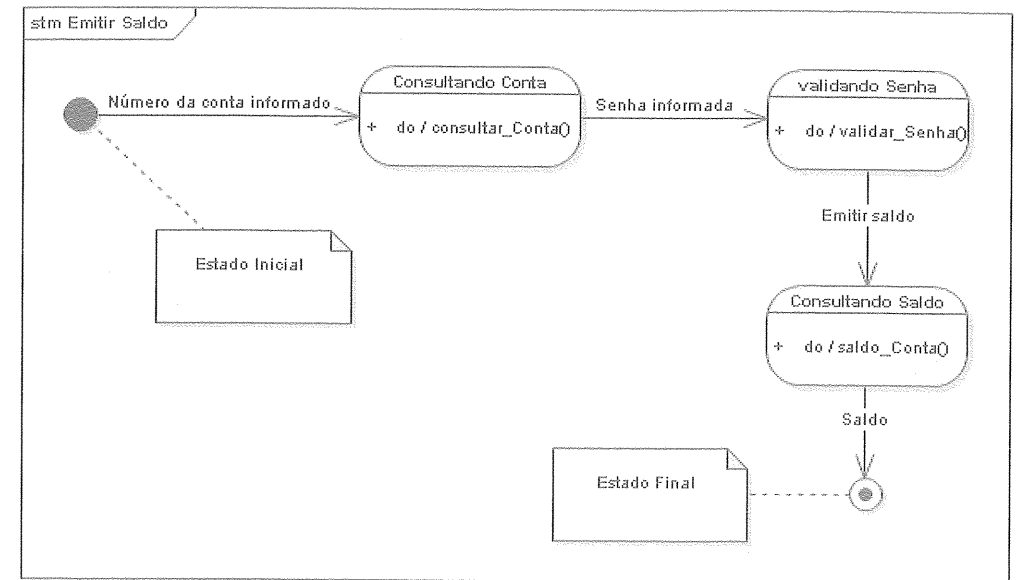


Figura 9.4 – Exemplo de Estado Inicial e Final – Processo de Emissão de Saldo.

Nesse exemplo, representamos somente os estados pelos quais passa um objeto da classe *Conta_Comum* durante o processo de emissão de saldo. Contudo, poderíamos detalhar melhor esse processo apresentando os estados do processo como um todo. Nessa situação, estaríamos modelando os estados do caso de uso *Emitir Saldo*, conforme demonstra a figura 9.5.

Aqui inserimos um estado após o estado *Consultando Conta*, onde a transição gerada determina que a conta pesquisada foi encontrada e que o processo deve passar ao estado *Solicitando Senha*, permanecendo nesse estado até que o cliente forneça uma senha, o que irá gerar o estado *Validando Senha*. Inserimos ainda o estado *Apresentando Saldo* após o estado *Consultando Saldo*. Esse último estado é produzido pela transição contendo o saldo consultado e se refere à apresentação do saldo consultado na interface.

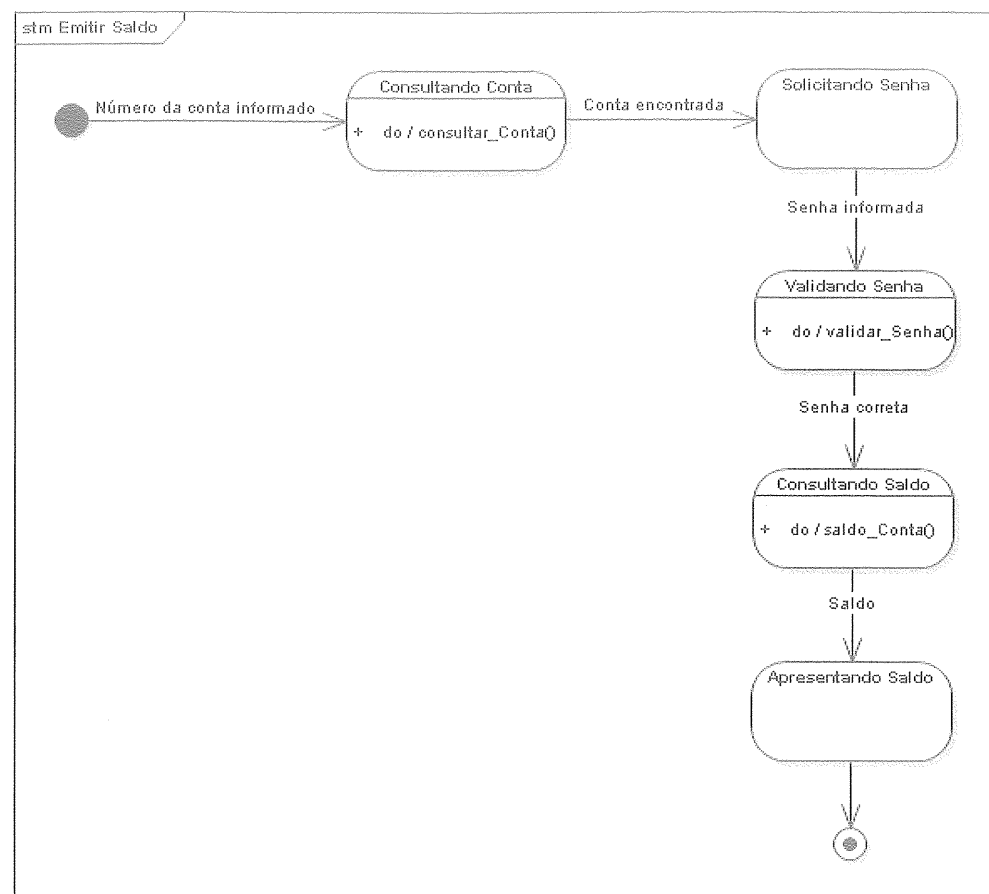


Figura 9.5 – Processo de Emissão de Saldo Detalhado.

9.5 Atividades internas

Ao observarmos a figura 9.5 perceberemos que alguns estados têm uma segunda divisão, além de seu título, e outros não, uma vez que não é obrigatório seu preenchimento. A segunda divisão representa as atividades que um objeto pode executar quando em um estado, conhecidas como atividades internas. Essas atividades podem ser detalhadas por meio das seguintes cláusulas:

- **Entry** – identifica uma atividade que é executada quando o objeto assume (entra em) um estado. Sempre que um estado é assumido, ele executa o comportamento identificado por essa cláusula antes de qualquer outra ação.
- **Exit** – identifica uma atividade que é executada quando o objeto sai de um estado. Sempre que se vai sair de um estado, o comportamento identificado por essa cláusula é executado antes de o estado ser abandonado.

- **Do** – identifica uma atividade realizada durante o tempo em que o objeto se encontra em um estado. Atividades internas do tipo Do também são chamadas de atividades de estado.

As cláusulas Entry e Exit, também chamadas ações de estado, estão mais associadas às transições do que ao estado propriamente dito, já que são executadas quando o objeto assume um novo estado ou quando está mudando de estado. Além disso, seu tempo de execução costuma ser inferior às atividades de estado, podendo representar a simples atribuição de um valor a um atributo ou a geração de uma saída, enquanto as atividades de estado normalmente representam métodos executados pelo objeto. É preciso destacar que essas cláusulas não são obrigatórias e nem sempre um estado conterá qualquer uma delas.

As atividades internas são representadas em uma segunda divisão do estado, conforme demonstra a figura 9.6, na qual um objeto está executando uma atividade, conforme mostra a cláusula Do.

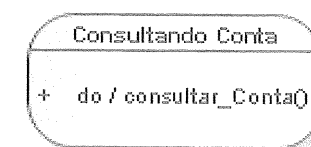


Figura 9.6 – Estado Representando a Execução de uma Atividade.

No exemplo apresentado na figura 9.6, enquanto o objeto estiver no estado **Consultando Conta**, ele executará o método `consultar_Conta`. Da mesma forma, ao observarmos a figura 9.5 percebemos que durante o estado **Validando Senha** será executado o método `validar_Senha` e que durante o estado **Consultando Saldo** será executado o método `saldo_Conta`. Já no estado **Solicitando Senha** não está sendo representada nenhuma atividade, uma vez que esse estado representa a espera para que o cliente informe a senha, enquanto o estado **Apresentando Saldo** representa a simples apresentação do valor do saldo na interface. Poderia-se, no entanto inserir uma cláusula Entry nesse último estado, informando que ao entrar nesse estado deveria ser apresentado o valor do saldo, mas acreditamos que isso seria um pouco redundante, já que o título já demonstra isso.

9.6 Transições Internas

Transições internas são aquelas que não produzem modificações no estado de um objeto. A figura 9.7 apresenta um exemplo de transição interna.



Figura 9.7 – Exemplo de Transição Interna.

Nesse exemplo enfocamos o estado **Registrando Pessoa**, cuja função é cadastrar um novo cliente da instituição bancária, representado por uma pessoa física. Nesse estado existem duas cláusulas: a primeira representa uma atividade de estado, conforme demonstra a cláusula **Do**, que determina que deverá ser executado o método `regPes` quando o objeto se encontrar neste estado. A segunda representa uma transição interna que determina que o CPF da pessoa deverá ser validado por meio da execução do método `valCpf`, que será chamado durante a execução do método `regPes`, ou seja, antes de concluir o registro da pessoa é necessário verificar se o CPF informado está correto. Assim, embora o objeto esteja validando o CPF informado pelo cliente, ele ainda se encontra no estado **Registrando Pessoa**, não havendo uma mudança no estado do objeto.

9.7 Autotransições

Autotransições apresentam pequenas diferenças em relação às transições internas. As transições internas ocorrem durante um estado do objeto, sem modificá-lo, enquanto as autotransições saem do estado atual do objeto, podendo executar alguma ação quando dessa saída, e retornam ao mesmo estado. Uma autotransição é representada por uma seta de transição que parte do objeto e retorna ao próprio objeto. A figura 9.8 apresenta um exemplo de autotransição.

Nesse exemplo enfocamos o processo de atendimento de um pedido. Pode acontecer de, ao atender um pedido, nem todos os itens solicitados estejam disponíveis em estoque, sendo preciso adquiri-los. Assim, sempre que um novo item for adquirido ocorre uma autotransição. No entanto, se algum item ainda estiver indisponível, o objeto volta ao estado **Atendendo Pedido**. Somente quando todos os itens do pedido estiverem disponíveis será possível passar ao estado seguinte. Observe que existem duas condições de guarda associadas às transições que determinam a condição para que se permaneça no estado **Atendendo Pedido** ou se passe para o estado **Finalizando Pedido**. O evento que causa a autotransição é identificado pela aquisição de um novo item. A esse evento existe uma condição de guarda associada, definida como “Nem todos os itens disponíveis”, o

que determina que se deve voltar ao mesmo estado quando essa condição for verdadeira. Já a transição que leva ao novo estado **Finalizando Pedido** tem como condição que todos os itens estejam disponíveis.

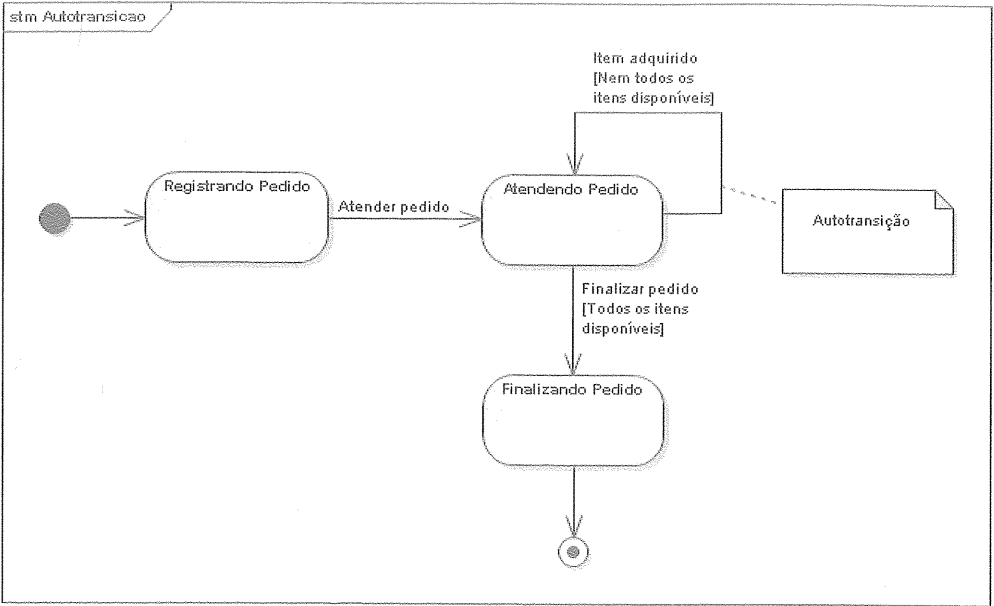


Figura 9.8 – Exemplo de Autotransição.

9.8 Pseudoestado de Escolha

Conhecido nas versões anteriores como estado de ponto de escolha dinâmico, o pseudoestado de escolha representa um ponto na transição de estados de um objeto em que deve ser tomada uma decisão, a partir da qual um determinado estado será ou não gerado, normalmente em detrimento de diversos outros possíveis estados. Assim, um pseudoestado de escolha representa uma decisão, apoiada por condições de guarda, em que se decidirá qual o próximo estado do objeto a ser gerado. Um pseudoestado de escolha pode ser representado por um losango ou por um círculo vazio de onde partem duas ou mais possíveis transições. A figura 9.9 apresenta um exemplo de pseudoestado de escolha enfocando o processo de manutenção do cadastro de clientes.

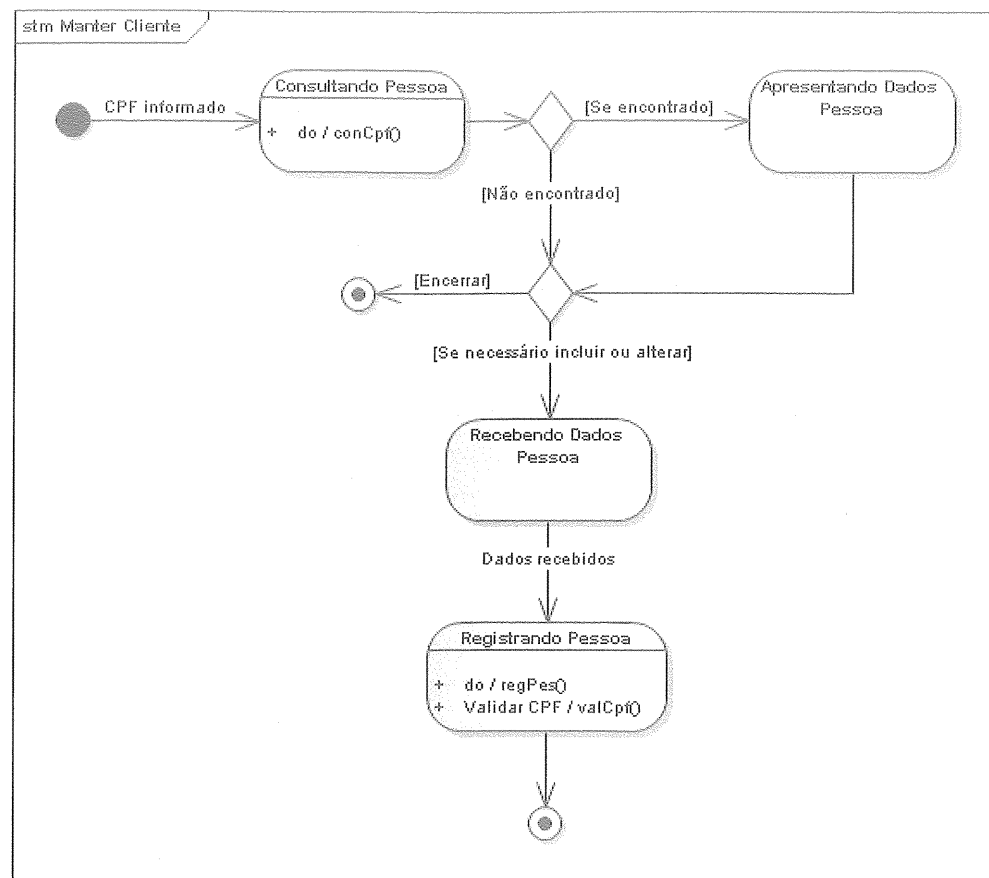


Figura 9.9 – Pseudoestado de Escolha – Manter Clientes.

Ao observarmos a máquina de estados apresentada no diagrama, percebemos que o processo inicia-se quando é informado o CPF da pessoa, o que gera o estado **Consultando Pessoa**, onde é executado o método `conCpf()`. Quando o estado é concluído é necessário fazer uma escolha, representada por um pseudoestado de escolha, onde, se for encontrada uma pessoa com o CPF informado, o processo deverá passar ao estado **Apresentando Dados Pessoa** (conforme demonstra a condição de guarda) e, em seguida passar a um segundo pseudoestado de escolha. Se a pessoa não for encontrada, então o estado **Apresentando Dados Pessoa** não é gerado e passa-se diretamente ao segundo pseudoestado de escolha, em que se deve escolher entre encerrar o processo ou fornecer os dados da pessoa para inserir uma nova pessoa ou alterar os dados da pessoa encontrada, o que produz o estado **Recebendo Dados Pessoa** e, no momento em que esse for concluído, passa-se ao estado **Registrando Pessoa**, no qual serão executados os métodos `regPes()` e `valCpf()`, já explicados.

9.9 Barra de Bifurcação/União

É utilizada quando da ocorrência de estados paralelos, causados por transições concorrentes. Sua função é determinar o momento em que o processo passou a ser executado em paralelo e em quantos subprocessos se dividiu (evento conhecido como bifurcação) ou determinar o momento em que dois ou mais subprocessos se uniram em um único processo (evento conhecido como união). Pode ser representada tanto por uma barra horizontal quanto uma vertical. A figura 9.10 apresenta um exemplo de barra de bifurcação/união, onde são modelados dois estados paralelos pelos quais passa um objeto da classe **carro** no momento em que um ator necessita dirigi-lo.

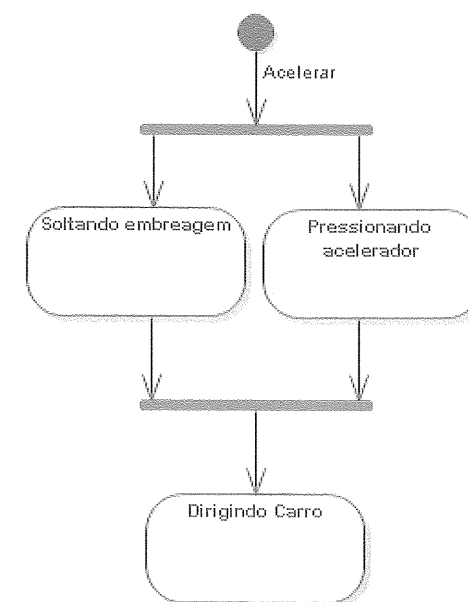


Figura 9.10 – Barra de Bifurcação/União.

Esse exemplo representa um trecho de um diagrama que modela os estados pelos quais passa um objeto da classe **Carro** no processo que o usuário leva para destrancá-lo, ligá-lo e dirigi-lo. No trecho ilustrado na figura, enfocamos o momento final do processo, quando o motorista começa a guiar o carro, mas, para isso, ele precisa realizar duas tarefas simultaneamente: soltar a embreagem e pressionar o acelerador. Dessa forma, foi inserida uma barra de bifurcação/união horizontal, indicando que os dois estados ocorrem em paralelo. Em seguida usa-se uma nova barra de bifurcação/união para unir os dois subprocessos e gerar o estado **Dirigindo Carro**.

Outro exemplo para o uso da barra de bifurcação/união pode ser visto na figura 9.11, onde enfocamos um processo que se refere a uma pesquisa internet onde, em paralelo com a busca por informações solicitadas são pesquisadas também ofertas relacionadas à pesquisa.

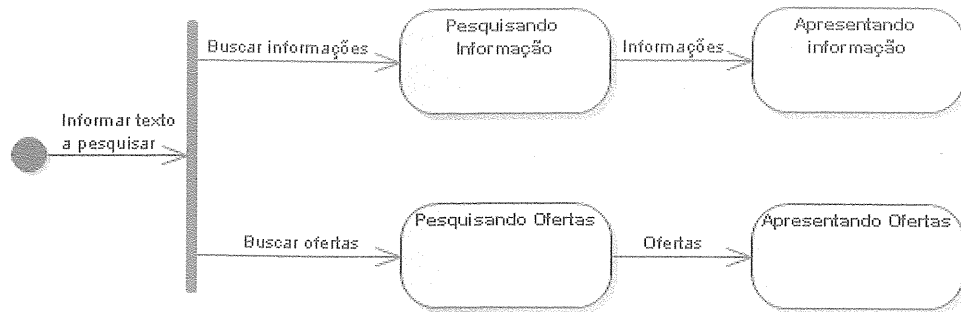


Figura 9.11 – Uso da barra de bifurcação/união no processo de Solicitar Pesquisa.

Nesse exemplo, no momento em que o evento “Informar texto a pesquisar” é disparado, uma barra de bifurcação/união divide os estados gerados pelo evento, havendo um fluxo para buscar e apresentar as informações pertinentes à pesquisa e outro para buscar e apresentar as ofertas relacionadas à mesma.

A figura 9.12 apresenta ainda um diagrama de máquina de estados completo que utiliza barras de bifurcação/união, referentes ao processo de abrir, ligar e dirigir um carro. Além de completar o exemplo apresentado na figura 9.10, esse diagrama demonstra também um bom uso para as cláusulas Entry e Exit.

Em seu estado inicial, o carro encontra-se fechado. A representação desse estado não era obrigatoriamente necessária, já que o diagrama poderia iniciar a partir do estado seguinte. A representação deste estado serve apenas para demonstrar o estado em que o carro se encontrava antes do início do processo, ou seja, o objeto estava estático, esperando por um evento.

O processo realmente inicia quando o motorista decide abrir o carro. Tal decisão gera uma transição, que, por sua vez, cria um novo estado, chamado aqui **Abrindo Carro**. Durante esse estado são executadas algumas tarefas sobre o objeto carro, detalhadas nas cláusulas Entry (Abrir porta), Do (Girar chave) e Exit (Retirar chave), representadas na segunda divisão do Estado.

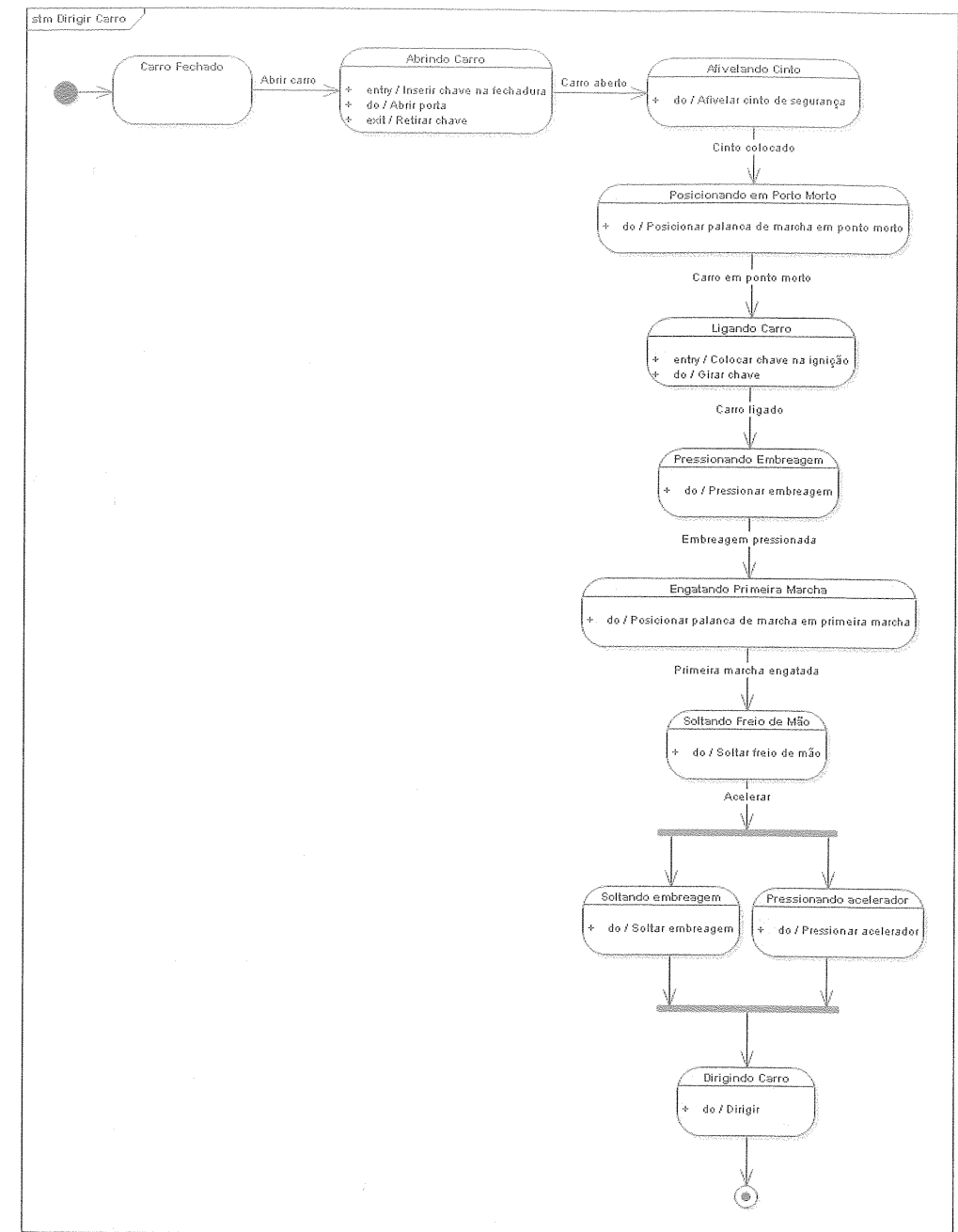


Figura 9.12 – Estados de um objeto carro durante o processo para dirigi-lo.

Quando o estado **Abrindo Carro** é iniciado, o motorista insere a chave na fechadura do carro para abri-lo. Como essa é uma ação realizada logo no início do estado, foi colocada em uma cláusula **Entry**, como pode ser observado na figura. Em seguida, o motorista gira a chave e abre o carro, o que representa a atividade principal do estado, razão pela qual foi colocada junto à cláusula **Do**, que armazena as atividades realizadas enquanto um objeto encontra-se em um determinado estado. Ao finalizar a abertura do carro, o motorista retira a chave. Como essa ação é realizada quando o estado é concluído, foi posta em uma cláusula **Exit**, que armazena as ações realizadas durante a finalização de um estado. Além disso, os atos de inserir e retirar a chave da fechadura do carro foram colocados nas cláusulas **Entry** e **Exit** por serem ações muito simples e rápidas, não representando verdadeiras atividades.

Estando o carro aberto, o motorista afivelará o cinto de segurança. Poderia-se afirmar que primeiro o motorista deveria entrar no carro. Em termos de algoritmo, estaria correto, contudo, nesse diagrama estamos enfocando somente os estados de um objeto da classe **Carro**, e não do motorista. Na realidade, até mesmo o estado **Afivelando Cinto** poderia ser considerado desnecessário no diagrama, por relacionar-se mais ao motorista do que ao carro propriamente dito. Entretanto, como é um estado que ocorre sobre o objeto carro, resolvemos também representá-lo.

Depois de ter afivelado o cinto de segurança, o motorista colocará o carro em ponto morto (ou neutro para alguns), ou seja, desengatará qualquer marcha em que o carro estivesse engatado. Esse ato irá gerar uma nova transição, que mudará o estado do objeto para **Colocando em Ponto Morto**. Durante esse estado, o motorista realizará a atividade de posicionar a palanca de marcha em ponto morto, assim, essa atividade é posicionada junto à cláusula **Do**.

Logo depois de desengatar o carro, o motorista irá ligá-lo. Ao iniciar essa operação, o objeto carro assumirá um novo estado denominado **Ligando Carro**. No começo desta operação, o motorista colocará a chave na ignição. Como se trata de uma ação simples e realizada no exato momento em que o objeto assume o estado enfocado, ela é posicionada junto à cláusula **Entry**. Logo depois de inserir a chave na ignição, o motorista irá girá-la. A atividade de girar a chave é a principal do estado **Ligando Carro**, por este motivo é colocada junto à cláusula **Do** desse estado.

Estando o carro ligado, o motorista irá pressionar a embreagem e engatar a primeira marcha, gerando os estados **Pressionando Embreagem** e **Engatando Primeira Marcha**, nessa ordem, como podemos observar. Esses dois estados executarão as atividades de pressionar embreagem e posicionar a palanca de marcha em primeira marcha.

No momento em que a primeira marcha estiver engatada, passa-se ao estado **Soltando Freio de Mão**, cuja atividade, descrita pela cláusula **Do**, é justamente soltar o freio de mão.

Para começar a dirigir o carro, o motorista terá de realizar duas tarefas simultâneas, as quais forçarão a geração de dois novos estados para o objeto, a saber: **Soltando Embreagem** e **Pressionando Acelerador**. Nesse caso, os dois estados ocorrem em paralelo, razão pela qual se faz necessário o uso de uma barra de bifurcação/união. Observe que utilizamos uma barra de bifurcação/união para separar o processo em dois e outra para unir os dois processos em um único novamente.

O resultado da execução desses dois estados gera o último estado representado no diagrama, no qual o motorista passa a dirigir efetivamente o veículo. Como este é um dos estados em que o objeto irá se manter por mais tempo, colocou-se a atividade dentro de uma cláusula **Do**, porque esta será a atividade constante do estado.

9.10 Estados Compostos

Estados compostos são o segundo tipo de estado suportado pelo diagrama de máquina de estados. O primeiro é o estado simples já explicado anteriormente neste capítulo, e o terceiro é a máquina de subestados que será explicada mais adiante.

Um estado composto é um estado que contém internamente dois ou mais estados, chamados subestados. Um subestado é chamado direto quando não está contido por outro estado ou indireto em caso contrário. Assim, um estado composto é um estado que foi “explorado”, de maneira a apresentar detalhadamente todas as etapas pelas quais passa o objeto quando no estado em questão.

Um estado composto pode apresentar somente uma região ou ser decomposto em duas ou mais regiões ortogonais, quando se torna um estado ortogonal (chamado estado concorrente em versões anteriores), onde cada região terá estados e transições. Cada região de um estado composto pode ter um pseudoestado inicial e um estado final. Uma transição para um estado composto representa uma transição para o pseudoestado inicial de cada região. Um estado composto não pode ser reutilizado, diferentemente do que ocorre com as máquinas de subestado.

Um estado composto pode facilitar a compreensão de um determinado estado de maneira bem mais detalhada, em que são discriminados os diversos subestados pelos quais passa o objeto quando no estado composto. A figura 9.13 demonstra um exemplo de estado composto com apenas uma região.

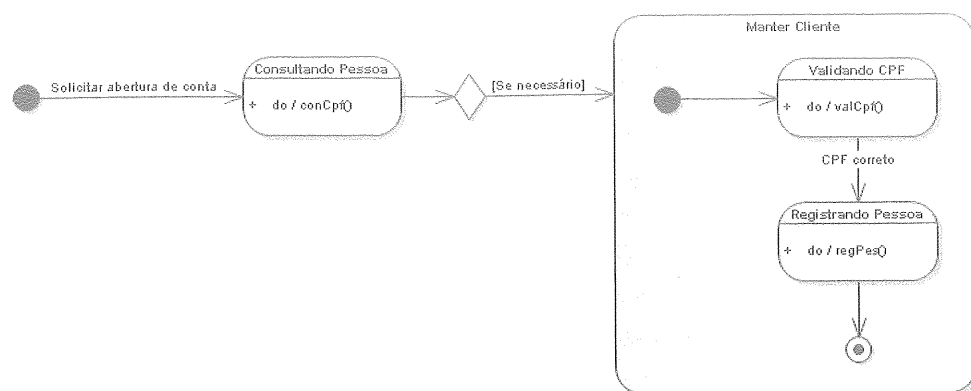


Figura 9.13 – Estado Composto.

No exemplo dessa figura, iniciamos a modelar o processo de abertura de conta no qual o evento “solicitar abertura de conta” produz uma transição que gera o estado **Consultando Pessoa**. Depois de esse estado ser concluído passa-se a um pseudoestado de escolha (somente identificamos uma de suas transições, uma vez que o exemplo está incompleto), onde, se for necessário (como demonstra a condição de guarda), gera-se uma transição para um estado composto intitulado **Manter Clientes**, que representa os estados necessários para dar manutenção no cadastro de pessoas, ou seja, para incluir ou alterar o registro de uma pessoa.

O leitor notará que transformamos o estado **Registrando Pessoa**, exemplificado anteriormente, em um estado composto, dividindo-o em dois subestados. Nesse caso, tomamos a transição interna para validação de CPF e a transformamos em um subestado, assim como criamos um segundo subestado, denominado **Registrando Pessoa**. Dessa forma, antes de concluir o cadastro ou atualização da pessoa, primeiramente valida-se seu CPF e, caso este esteja realmente correto, registra-se os dados da pessoa.

O estado inicial apresentado dentro do estado composto na verdade é um pseudoestado inicial, já que o diagrama já tem um estado inicial. Esse pseudoestado inicial refere-se somente ao início dos subestados do estado composto.

9.11 Pseudoestado de História

Um pseudoestado de história representa o registro do último subestado em que um objeto se encontrava, quando, por algum motivo, o processo foi interrompido. Assim, por meio do pseudoestado de história, podemos retornar exatamente ao último subestado em que o objeto encontrava-se quando da interrupção do processo. Um pseudoestado de história é representado por um H dentro de um círculo, conforme mostra a figura 9.14.

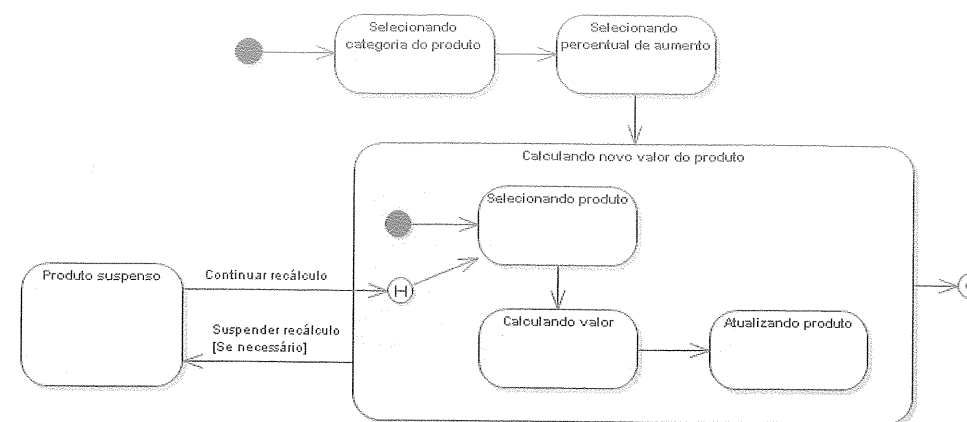


Figura 9.14 – Pseudoestado de História.

Nesse exemplo, enfocamos um processo de aumento de produtos de uma determinada categoria. Primeiramente, selecionamos a categoria do produto e depois o percentual de aumento para os produtos da categoria escolhida. O recálculo de valor de todos os produtos da categoria em questão é representado por um estado composto, onde um produto específico da categoria é selecionado, seu valor de venda é recalculado, e o produto, atualizado. Observe que um estado de história mantém o subestado atual do processo.

Eventualmente, pode acontecer que, por um motivo qualquer, seja necessário suspender temporariamente o processo, conforme demonstra a transição à esquerda do estado composto, que gera o estado **Produto suspenso**. Tão logo o motivo da interrupção seja sanado, o processo de recálculo pode ser retomado a partir do momento em que foi suspenso, por meio do estado de história. Observe que é gerada uma transição do estado **Produto suspenso** diretamente para o estado de história, que será responsável por continuar o processo exatamente a partir do subestado onde havia sido interrompido.

Existe também o pseudoestado de história profunda, representado por um H seguido de um símbolo de asterisco (H*) envolvido por um círculo, utilizado quando da ocorrência de estados compostos dentro de estados compostos. Assim, o estado de história profunda guarda e recupera o último subestado em qualquer dos estados compostos onde ele possa estar.

9.12 Estados Compostos Ortogonais

Um estado ortogonal (chamado concorrente em versões anteriores) é um estado composto que possui mais de uma região, onde cada região apresenta um conjunto de estados e os estados de cada região são assumidos paralelamente,

o que força o processo a se dividir em dois ou mais subprocessos concorrentes. Os processos concorrentes são separados por uma linha tracejada dentro do estado ortogonal, conforme pode ser observado na figura 9.15.

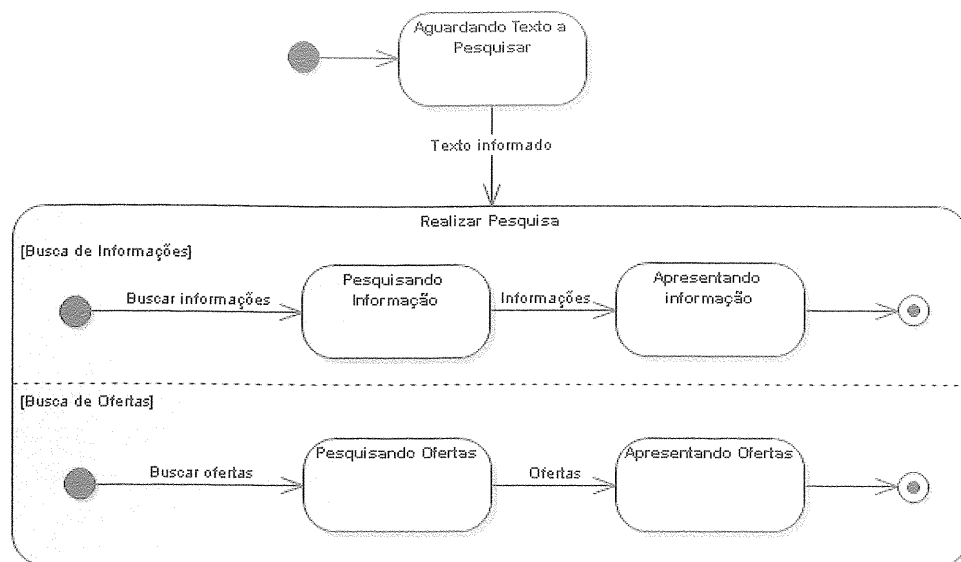


Figura 9.15 – Exemplo de Estado Composto Ortogonal.

A figura 9.15 apresenta uma segunda alternativa para o processo de Solicitar Pesquisa. Nesse exemplo, o processo se inicia com um estado estático que aguarda que o usuário digite um texto a pesquisar. O evento de inserção de um texto para pesquisa causa uma transição para um estado composto ortogonal contendo duas regiões. Na primeira região, como seu título na forma de condição de guarda informa, é feita a busca por informações relevantes à pesquisa, enquanto na segunda região é feita a busca por ofertas relacionadas ao texto pesquisado. Os estados de cada região ocorrem paralelamente.

9.13 Estado de Sincronismo

Em alguns processos pode eventualmente ser necessário que estados de regiões diferentes estejam de alguma forma sincronizados, por vezes sendo necessário que um estado de uma região espere por um estado de outra. Assim, é preciso existir um estado de sincronismo, cuja função é permitir que os relógios de duas ou mais regiões estejam sincronizados em um determinado momento do processo. O estado de sincronismo é representado por um símbolo de asterisco (*) dentro de um círculo, conforme pode ser observado na figura 9.16.

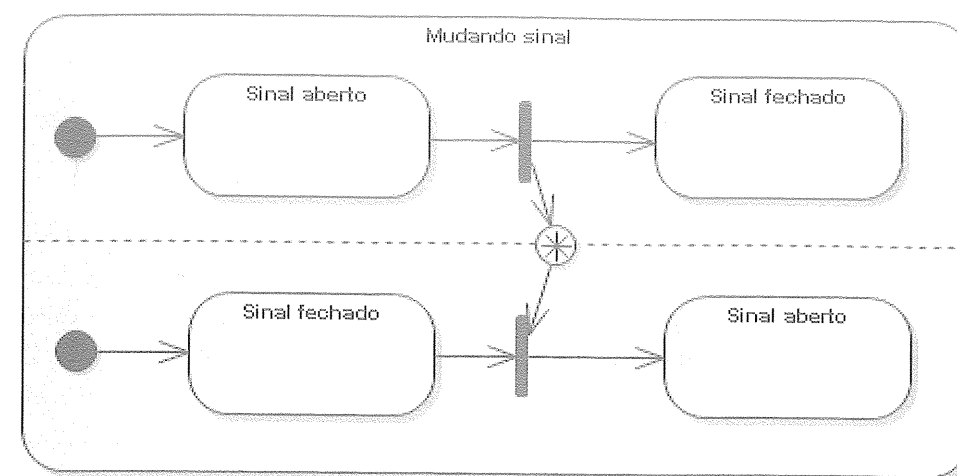


Figura 9.16 – Estado de Sincronismo.

Nesse exemplo há dois sinais de trânsito. No momento em que o sinal de um muda, o outro deve mudar também, automaticamente. Assim, o estado do primeiro sinal de trânsito é representado como **Sinal Aberto**, e o do segundo, como **Sinal Fechado**. Quando o primeiro sinal de trânsito recebe a ordem de trocar seu sinal, este gera duas transições, uma para trocar seu próprio estado e outra para avisar ao estado de sincronismo para alterar também o estado do segundo sinal. Assim, no momento em que o estado do primeiro sinal mudar para **Sinal Fechado**, o estado do segundo sinal automaticamente terá de mudar para **Sinal Aberto**.

9.14 Estado de Submáquina

Uma estado de submáquina é um mecanismo de decomposição que permite a fatoração de comportamentos comuns e seu reuso. Um estado de submáquina é equivalente a um estado composto. No entanto, seus subestados não são descritos no diagrama, o que indica que terão de ser demonstrados em outro diagrama. Além disso, diferente de um estado de submáquina, um estado composto não pode ser reutilizado. Um estado de submáquina é representado por um retângulo com as bordas arredondadas sem divisões internas, contendo em seu canto inferior direito um símbolo que representa um diagrama de máquina de estados, significando que o estado em questão possui subestados internos. A figura 9.17 apresenta um exemplo de estado de submáquina.

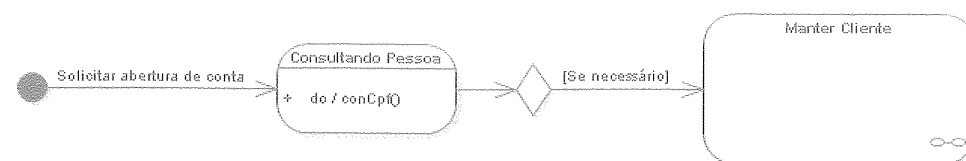


Figura 9.17 – Estado de Submáquina.

Aqui utilizamos novamente o exemplo parcial do processo de **Abertura de Conta** já apresentado, e substituímos o estado composto denominado **Manter Cliente** para um estado de submáquina de mesmo nome, em que os subestados não são representados, mas deverão estar detalhados em outro diagrama de máquina de estados, normalmente com o nome do próprio estado de submáquina. Essa abordagem permite produzir diagramas mais enxutos, deixando os diagramas mais fáceis de entender, além de permitir referenciar outros diagramas já modelados.

9.15 Pseudoestado de Junção

Esse pseudoestado é utilizado para projetar caminhos transacionais complexos, podendo unir múltiplos fluxos em um único ou dividir um fluxo em diversos, podendo utilizar condições de guarda como auxílio. Um pseudoestado de junção é representado por um símbolo idêntico ao de estado inicial, ou seja, um círculo preenchido. A figura 9.18 apresenta um exemplo de pseudoestado de junção utilizado durante o processo de abertura de conta.

Aqui apresentamos o diagrama de máquina de estados equivalente ao processo de abertura de conta, que iniciamos a modelar nos exemplos anteriores. Observe que, depois de realizar o teste para determinar se é necessário efetuar manutenção no cadastro de clientes, o fluxo foi dividido em dois, sendo novamente unido por um pseudoestado de junção, identificado aqui por uma nota. Após a união dos fluxos gera-se uma transição para o estado **Abrindo Conta**, onde é executado o método **abrir_Conta** e, depois de sua conclusão, gera-se uma transição para registrar o depósito inicial. Observe que essa transição aponta para um estado de submáquina onde estão detalhados os estados do processo de **Realizar Depósito**. Após a conclusão desse estado de submáquina o processo é encerrado.

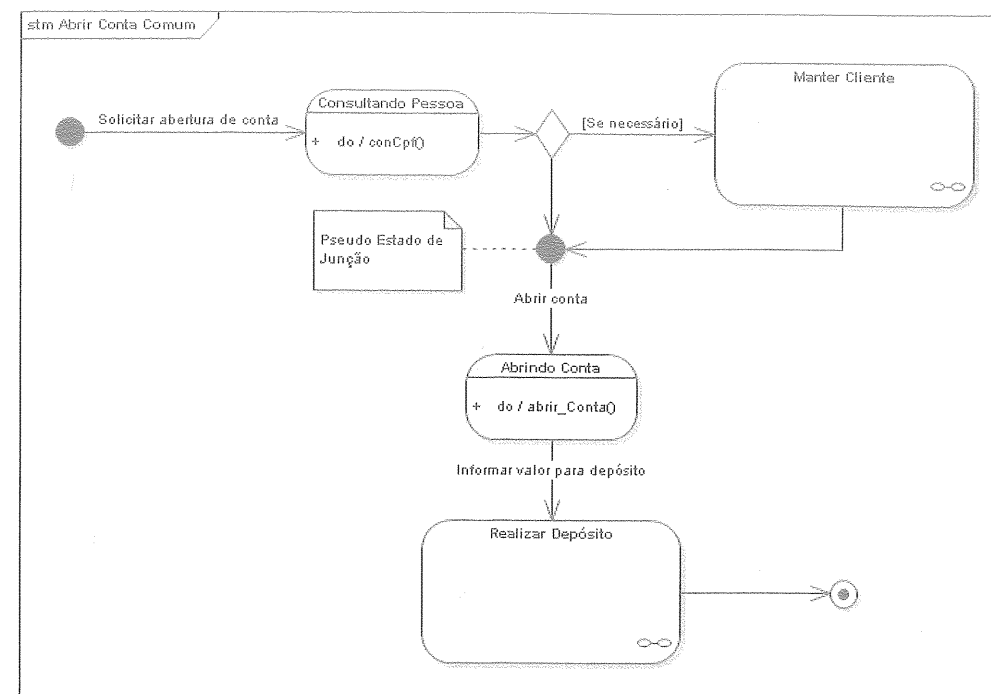


Figura 9.18 – Pseudoestado de Junção – Processo de Abertura de Conta.

9.16 Pseudoestado de Ponto de Entrada e Pseudoestado de Ponto de Saída

Os pseudoestados de ponto de entrada e de ponto de saída são utilizados juntamente com estados de submáquina ou estados compostos. Demonstram pontos de entrada e saída que serão somente usados em casos excepcionais. O estado de entrada demonstra um caminho alternativo e é representado por um círculo vazio na borda do estado de submáquina. A figura 9.19 apresenta um exemplo de pseudoestado de ponto de entrada.

Nesse exemplo, o processo normal executa primeiro o estado denominado **Iniciando** para depois executar a atividade representada pelo estado de submáquina. No entanto, pode acontecer de não haver necessidade de executar o estado preliminar. Dessa forma, foi inserido no estado de submáquina um pseudoestado de entrada, chamado **Pular Iniciação**, por meio do qual pode-se executar a atividade sem passar pela primeira etapa.

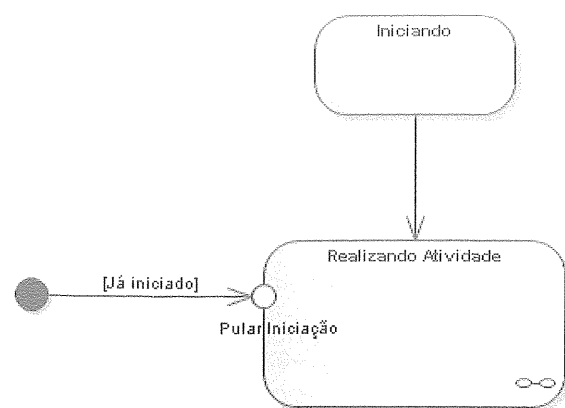


Figura 9.19 – Pseudoestado de Ponto de Entrada.

O estado de saída normalmente demonstra uma exceção que causa o cancelamento do fluxo normal estado. É representado por um círculo com um X. A figura 9.20 apresenta um exemplo de pseudoestado de ponto de saída.

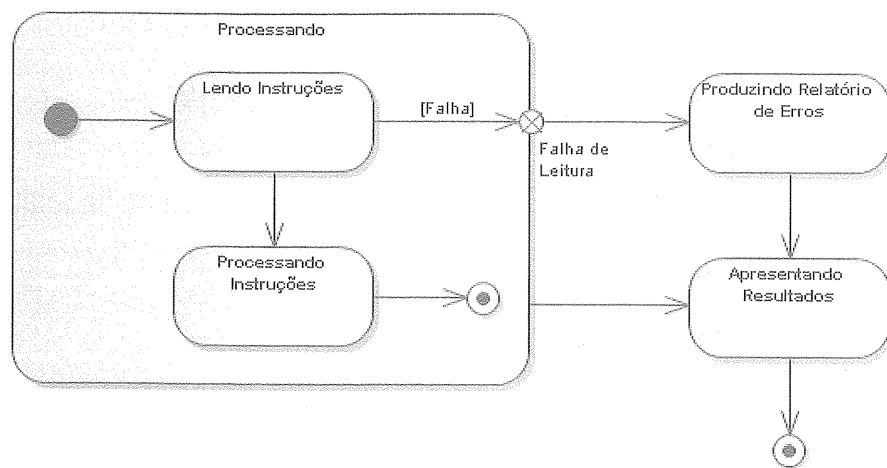


Figura 9.20 – Pseudoestado de Ponto de Saída.

Aqui representamos um estado composto responsável por ler e processar instruções. Após sua conclusão, passa-se ao estado **Apresentando Resultados**. Porém, durante a leitura das instruções pode ocorrer uma exceção, denotando uma falha de leitura das instruções. Essa exceção interrompe o processamento das instruções e gera um relatório de erros. Isso é representado por um pseudoestado de ponto de saída, chamado de **Falha de Leitura**.

Esses estados normalmente representam exceções, motivo pelo qual os exemplos apresentam também as transições normais que não necessitam utilizar esses pseudoestados ou as entradas e saídas alternativas por eles representados.

9.17 Pseudoestado de Término

Força o término da execução de uma máquina de estados, devido, por exemplo, à ocorrência de uma exceção. É representado por um "X", conforme demonstra a figura 9.21.

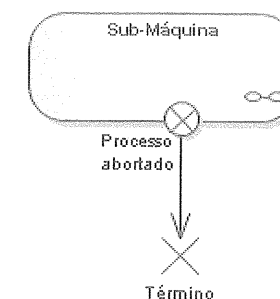


Figura 9.21 – Pseudoestado de Término.

9.18 Exemplo de Diagrama de Máquina de Estados – Emitir Extrato

Nesta seção apresentaremos os estados relativos ao caso de uso **Emitir Extrato** do sistema de controle bancário que vimos modelando ao longo do livro, conforme demonstra a figura 9.22.

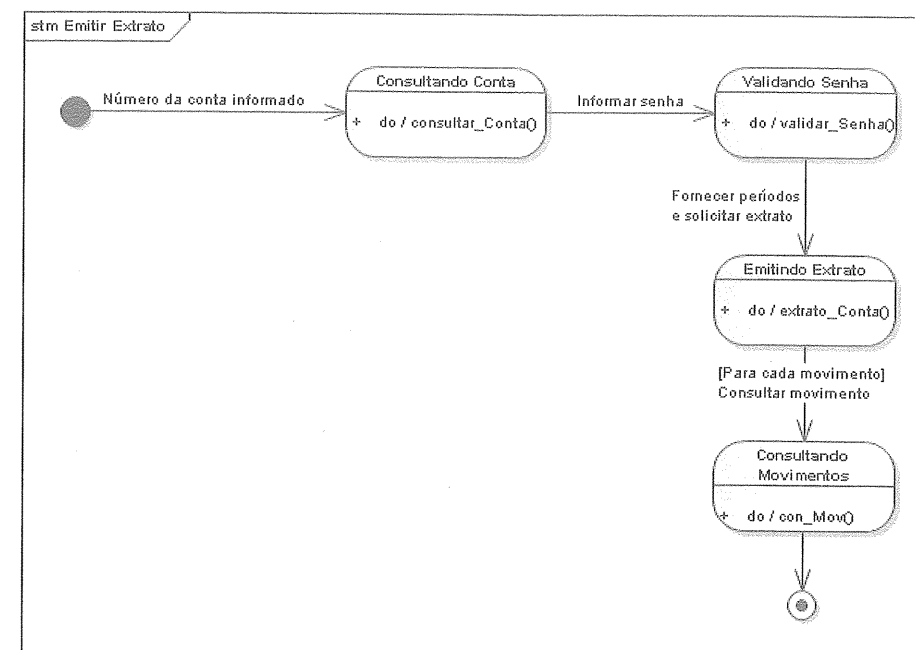


Figura 9.22 – Processo de Emissão de Extrato.

O processo se inicia quando a transição contendo o número da conta a consultar é disparada produzindo o estado Consultando Conta, no qual é executado o método consultar_Conta. O estado seguinte, Validando Senha, é gerado quando a senha da conta é informada. Nesse estado é executado o método validar_Senha. O evento no qual os períodos desejados para o extrato são informados gera o estado Emitindo Extrato, no qual será executado o método extrato_Conta. A partir desse estado gera-se uma transição onde serão consultados todos os movimentos do período relativos à conta consultada, por meio do estado Consultando Movimentos, onde é executado o método con_Mov.

9.19 Exemplo de Diagrama de Máquina de Estados – Realizar Depósito

Nesta seção serão apresentados os estados relativos ao caso de uso Realizar Depósito do sistema de controle bancário, por meio da figura 9.23.

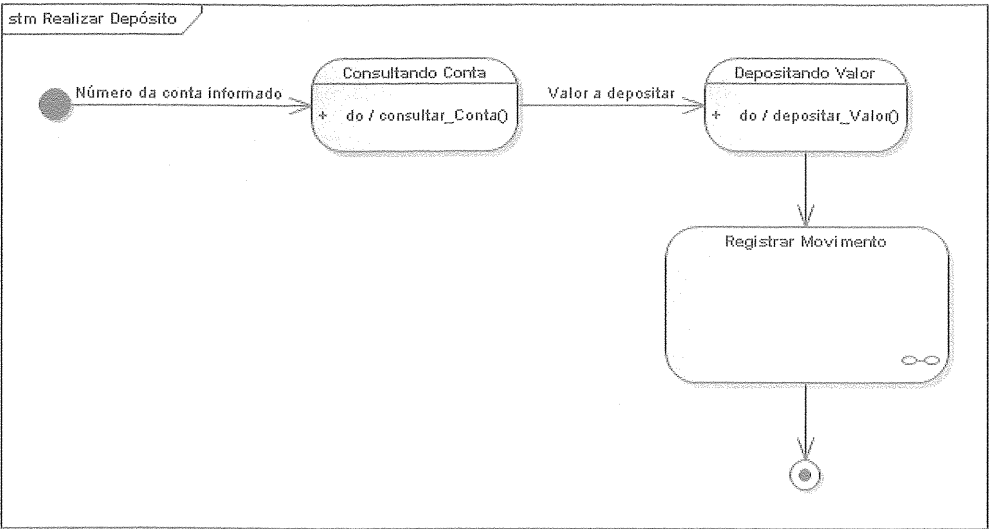


Figura 9.23 – Processo de Realizar Depósito.

Esse diagrama se inicia quando o cliente fornece o número da conta para a qual deseja depositar um valor, o que dispara o método consultar_Conta. Depois de a conta ter sido consultada, o evento que representa a informação do valor a depositar gera o estado Depositando Valor, onde é executado o método depositar_Valor. A conclusão desse estado produz uma transição que leva a um estado de submáquina que representa o processo de Registrar Movimento. Depois de o registro do movimento ter sido concluído, o processo de Realizar Depósito é terminado.

9.20 Exemplo de Diagrama de Máquina de Estados – Realizar Saque

Já nesta seção demonstraremos os estados relativos ao caso de uso Realizar Saque, apresentados pela figura 9.24.

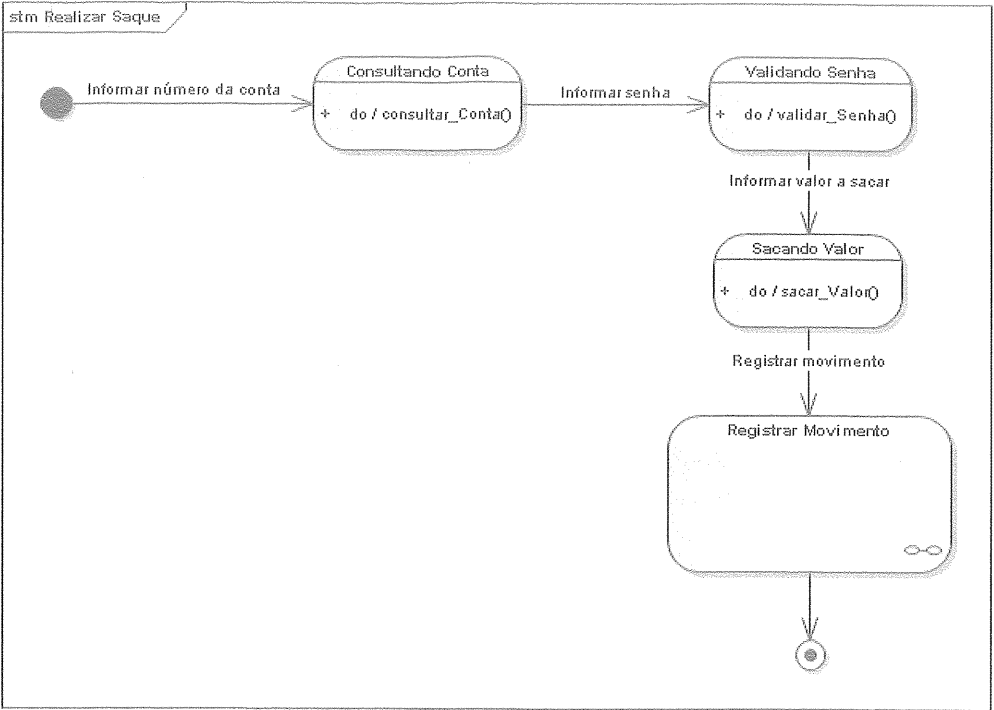


Figura 9.24 – Processo de Realizar Saque.

Novamente, o processo se inicia com a informação do número da conta que gera o estado Consultando Conta. Após este ter sido concluído, o evento que representa o fornecimento da senha produz o estado Validando Senha, em que é executado o método validar_Senha. A seguir, uma transição é gerada quando é informado o valor desejado para saque, que gera o estado Sacando Valor, onde é chamado o método Sacar_Valor. A retirada de um valor exige o registro do movimento, por isso, após a conclusão do estado, uma transição atinge um estado de submáquina que representa o processo de Registrar Movimento e, após seu término, o diagrama é concluído.

9.21 Exemplo de Diagrama de Máquina de Estados – Encerrar Conta

Finalmente, nesta seção apresentaremos os estados pertencentes ao caso de uso **Encerrar Conta** do sistema de controle bancário, como demonstra a figura 9.25. Esse diagrama apresenta uma complexidade maior em relação aos anteriores.

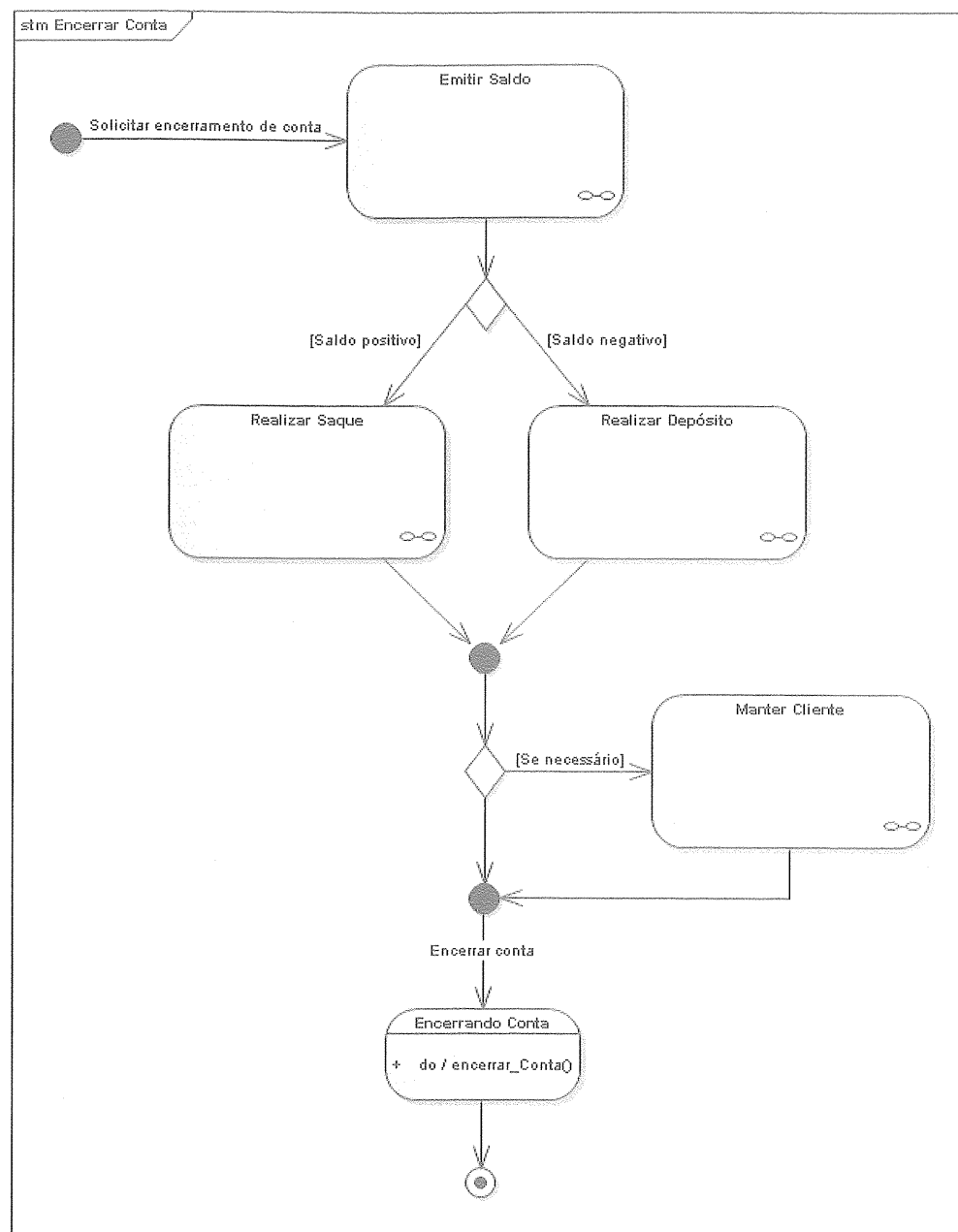


Figura 9.25 – Processo de Encerrar Conta.

Uma vez que diversos processos mais simples já haviam sido modelados anteriormente, lançamos mão de estados de submáquina para representá-los e deixar esse diagrama menor. Assim, o processo de encerramento de conta se inicia com o evento de solicitação de encerramento de conta, mas a transição que o representa atinge um estado de submáquina relacionado ao processo de **Emitir Saldo**, uma vez que é necessário primeiro saber quanto se tem na conta antes de encerrá-la. A partir desse estado de submáquina gera-se uma transição para um pseudoestado de escolha, onde é preciso escolher entre realizar um saque ou um depósito, dependendo de se o saldo estiver positivo ou negativo.

Assim, se o saldo estiver positivo, gera-se uma transição para o estado de submáquina **Realizar Saque**. Já se o saldo estiver negativo, gera-se uma transição para o estado de submáquina **Realizar Depósito**. Observe que o fluxo dividido pelo pseudoestado de escolha é unido novamente por um pseudoestado de junção e, a partir dele, é produzida uma nova transição para outro pseudoestado de escolha, onde se deve decidir se é necessário manter o cadastro do cliente, caso a conta a encerrar seja a única possuída por ele. Note que novamente o fluxo dividido é unido por um novo pseudoestado de junção e que, a partir deste, é produzida uma transição para o estado **Encerrando Conta**, no qual é executado o método `encerrar_conta`. A conclusão desse estado também conclui o diagrama.

Poderíamos, no lugar dos estados de submáquina, ter usado estados compostos, pois estes são equivalentes às máquinas de subestados, exceto por detalharem seus subestados e não poderem ser reutilizados. Porém, isso deixaria o diagrama bem mais extenso. Poderia-se também detalhar todo esse processo por meio de estados simples, se o projetista assim o preferisse.

9.22 Exercícios Propostos

Como já tem ocorrido nos capítulos anteriores, continuaremos a modelagem dos sistemas propostos pelos exercícios, enfocando agora o diagrama de máquina de estados.

9.22.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Desenvolva o diagrama de máquina de estados referente ao processo de venda de ingressos para um sistema de controle de cinema, sabendo que:

- Ao selecionar a opção de venda de ingressos, o sistema deverá apresentar todas as sessões ainda não encerradas. Cada sessão deve informar o título do filme e a sala em que será apresentado.

- A partir da listagem apresentada, o funcionário deverá escolher a sessão desejada pelo cliente.
- Finalmente, o funcionário deverá gerar o ingresso referente à sessão escolhida.

9.22.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

Desenvolva o diagrama de máquina de estados referente ao processo de pagamento de mensalidade para um sistema de clube social, levando em consideração os seguintes fatos:

- Primeiramente deve-se consultar o sócio que deseja pagar mensalidades.
- Após a consulta do sócio, deve-se consultar a(s) mensalidade(s) por ele devida(s).
- Se houver alguma mensalidade em atraso, deve-se calcular os juros referentes ao atraso do pagamento.
- Finalmente, deve-se quitar a(s) mensalidade(s).

9.22.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Desenvolva o diagrama de máquina de estados referente ao processo de locação de veículo para um sistema de aluguel de veículos, considerando que:

- Ao selecionar a opção de locação de veículos, o sistema deve carregar todos os clientes registrados.
- Em seguida, o sistema deve apresentar todos os veículos disponíveis. A listagem decorrente deve mostrar a descrição do automóvel, seu modelo e marca.
- A partir dessa listagem o funcionário deve selecionar o cliente.
- Depois de o cliente ter sido selecionado, deve-se selecionar o automóvel.
- Finalmente, depois de selecionar o veículo, o funcionário poderá mandar gerar a locação do automóvel selecionado.

9.22.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

Desenvolva o diagrama de máquina de estados referente ao processo de realizar leilão para um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- Ao receber a solicitação do serviço de realizar leilões, o sistema deve carregar todos os leilões ainda não encerrados na interface.
- A partir dessa listagem, o leiloeiro deve selecionar qual leilão deseja iniciar.

- No momento que um leilão for escolhido para ser iniciado, o sistema precisa carregar todos os itens a serem leiloados nele.
- A partir da listagem dos itens a serem leiloados, o leiloeiro deve escolher um item a leiloar e anunciá-lo.
- Se houver algum lance para o item anunciado, o sistema deve anunciá-lo e, em seguida, registrá-lo.
- Existe um tempo máximo de espera para que haja lances. Enquanto esse tempo não for atingido, o item permanece sendo anunciado.
- Quando o tempo máximo de espera por um lance for atingido, o processo deve verificar se houve ofertas para o item, caso em que se deve anunciar o participante que ofereceu o maior lance como vencedor. Caso contrário, deve-se simplesmente encerrar o anúncio do item.
- Depois de ter sido encerrado o anúncio de um item, deve-se verificar se ainda há itens a anunciar, caso em que o processo passa a anunciar o novo item. Caso contrário, o leilão deve ser encerrado.

9.22.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

Desenvolva o diagrama de máquina de estados referente ao processo de pagamento de diárias para um sistema de controle de hotelaria, de acordo com as seguintes definições:

- No momento em que o hóspede informa o número do quarto para quitar as diárias, o sistema deve consultar o hóspede e todas as diárias devidas pelo aluguel do quarto, apresentando-as ao funcionário.
- A partir dessa listagem, deve-se quitar as diárias apresentadas.
- Se tiver havido a solicitação de qualquer serviço no período em que o quarto estava ocupado, estes devem ser quitados também.
- O mesmo ocorre se houver quaisquer consumos de frigobar, sendo necessário também quitá-los.

9.23 Solução dos Exercícios

9.23.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

A figura 9.26 apresenta a solução para esse exercício. A seguir explicaremos cada um de seus estados.

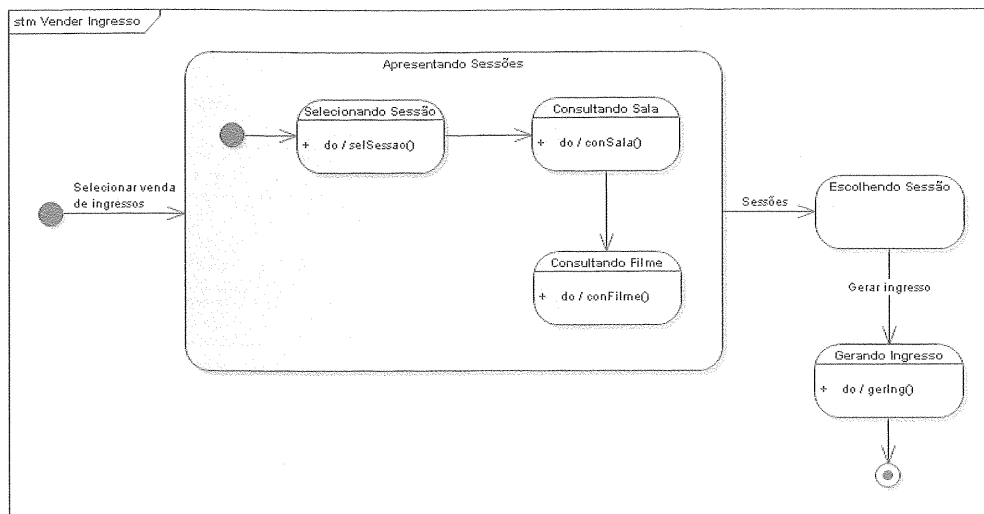


Figura 9.26 – Processo de Venda de Ingressos.

Esse processo se inicia com o funcionário escolhendo a opção de venda de ingressos. Essa escolha gera uma transição para o estado composto **Apresentando Sessões**, que contém três subestados. No primeiro, é selecionada uma sessão ainda não encerrada, onde é executado o método `selSessao`. O subestado seguinte consulta a sala onde ocorrerá a sessão, chamando o método `conSala`, e o terceiro subestado consulta o filme que será apresentado, executando o método `conFilme`.

Ao finalizar a consulta de sessões disponíveis é gerada uma transição para um estado de espera, onde o funcionário deverá escolher uma sessão. A escolha de uma sessão causa a transição para o estado **Gerando Ingresso**, no qual será disparado o método `gerIng` e, após a conclusão desse estado, o processo será encerrado.

9.23.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

A seguir, por meio da figura 9.27, apresentamos a solução para esse exercício.

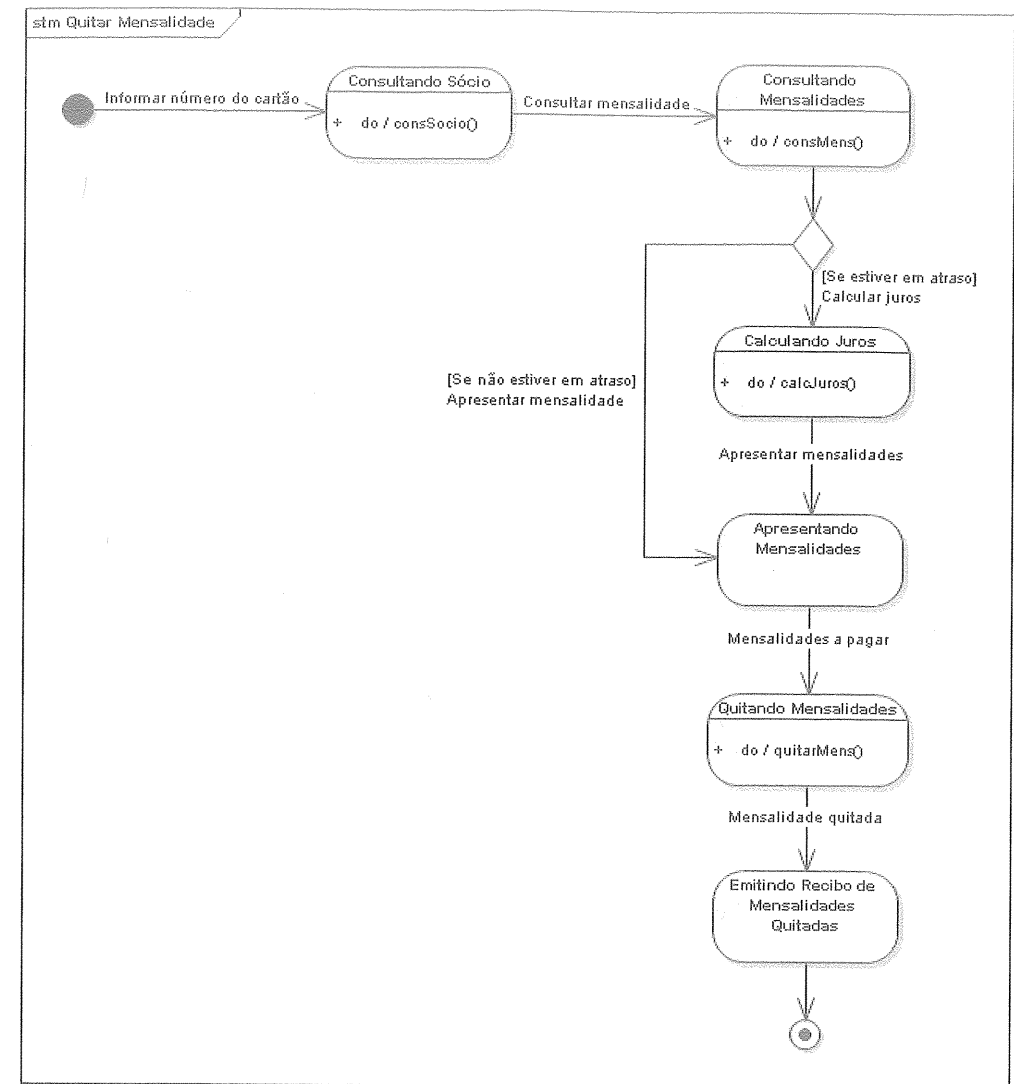


Figura 9.27 – Processo de Pagamento de Mensalidade.

Esse processo se inicia com o sócio fornecendo o número de seu cartão, o que gera o estado **Consultando Sócio** e a execução do método `consSocio`. Depois de consultar o sócio, uma transição passa ao estado **Consultando Mensalidades**, onde será chamado o método `consMens`.

Ao finalizar a consulta das mensalidades a pagar do sócio, uma nova transição atinge um pseudoestado de escolha onde deverá ser escolhida uma entre duas transições possíveis. Caso o sócio não tenha mensalidades em atraso, uma transição passa ao estado **Apresentando Mensalidades**, que simplesmente apresenta na interface o valor da mensalidade a ser paga. Caso o sócio tenha mensalidades

em atraso, uma transição gera o estado **Calculando Juros**, onde é disparado o método `calcJuros`, e após o término desse método, passa-se ao estado **Apresentando Mensalidades** já explicado.

A partir do estado **Apresentando Mensalidades** o sócio deverá escolher quais mensalidades deseja pagar. Essa escolha gera uma nova transição, que produz o estado **Quitando Mensalidades**, onde é executado o método `quitarMens`. O término desse estado se caracteriza por uma transição para o estado **Emitindo Recibo de Mensalidades Quitadas**, após o que o processo se encerra.

9.23.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Apresentaremos os estados da solução deste exercício, ilustrado pela figura 9.28, a seguir.

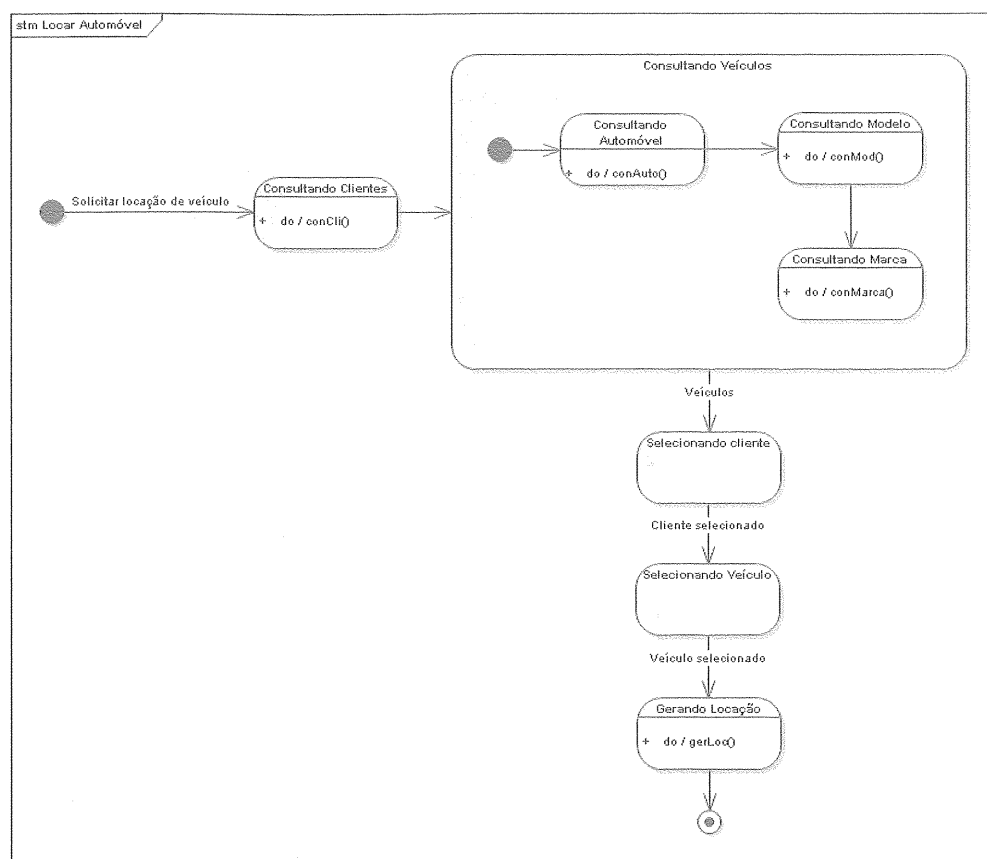


Figura 9.28 – Processo de Locação de Veículo.

Esse processo inicia-se com o funcionário solicitando o serviço de locação de veículos, o que causa uma transição para o estado **Consultando Clientes**, que carrega os clientes registrados por meio da execução do método `conCli`. Após a conclusão desse estado, passa-se ao estado composto **Consultando Veículos**, para carregar os veículos disponíveis. Esse estado composto apresenta três subestados. No primeiro é consultado um automóvel e executado o método `conAuto`, no segundo é consultado o modelo do automóvel pesquisado no subestado anterior, onde é executado o método `conMod` e, finalmente, no terceiro subestado, é consultada a marca do automóvel e executado o método `conMarca`.

Ao ser finalizado, esse estado composto gera uma transição contendo os veículos disponíveis e passa-se ao estado **Selecionando Cliente**, onde o processo aguarda que o funcionário escolha um cliente. A escolha de um cliente gera uma transição para o estado **Selecionando Veículo**, onde o processo aguarda que seja selecionado um dos automóveis disponíveis. A seleção de um automóvel gera uma transição para o estado **Gerando Locação**, onde o veículo é locado por meio da execução do método `gerLoc`.

9.23.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

A solução para esse exercício está contida na figura 9.29, que será explicada a seguir.

Esse processo inicia-se com o leiloeiro solicitando a execução do serviço de realizar leilões. Isso causa uma transição que leva ao estado **Consultando Leilões**, que tem por função apresentar os leilões ainda não encerrados, executando o método `conLeilao`. A partir dessa listagem o leiloeiro deve selecionar qual leilão deseja iniciar, o que gera o estado **Iniciando Leilao**, onde é chamado o método `iniciaLeilao` e, após a conclusão desse estado, é gerada uma transição para o estado **Consultando Itens**, cujo objetivo é carregar os itens a serem leiloados, no qual executa-se o método `conItem`.

A partir da listagem dos itens a serem leiloados o leiloeiro deve escolher um item a leiloar e anunciá-lo, o que é representado por um estado composto chamado **Leiloando Item**. O primeiro subestado desse estado composto representa o anúncio de um item a ser leiloadado. Durante esse estado, a página do leilão será atualizada para mostrar o referido item. A partir desse subestado passa-se a um pseudoestado de escolha, onde deve haver um teste. Se houver algum lance para o item anunciado, então passa-se ao subestado que representa o anúncio dessa oferta e em seguida para o subestado que representa o registro da mesma, onde executa-se o método `regLance`. Observe que o fluxo dividido pelo pseudoestado de escolha é unido novamente por um pseudoestado de junção.

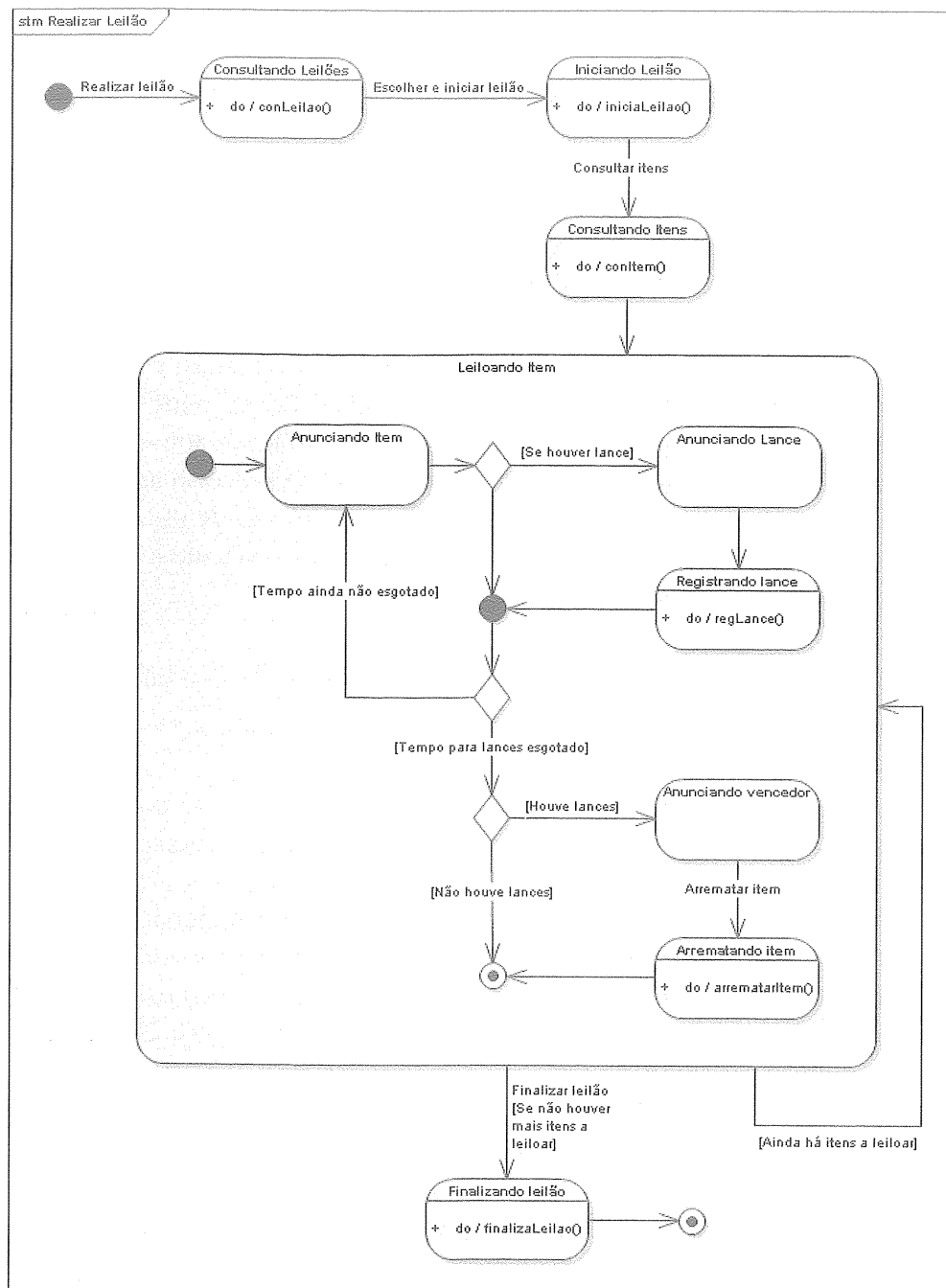


Figura 9.29 – Processo de Realizar Leilão.

O pseudoestado de junção gera uma transição para um segundo pseudoestado de escolha, onde é preciso decidir se o tempo máximo de espera para que haja lances para o item foi atingido ou não. Se o tempo máximo não foi atingido, o processo volta ao estado onde o item está sendo anunciado e continua aguardando lances. Caso contrário, passa-se a um terceiro pseudoestado de escolha, onde se deve verificar se houve ofertas para o item, caso em que gera-se uma transição para o subestado **Anunciando Vencedor** e, em seguida, para o subestado **Arrematando Item**, onde é executado o método `arrematarItem`, encerrando-se os subestados do estado composto. Se não tiver ocorrido nenhuma oferta, o estado composto é simplesmente encerrado.

A conclusão do estado composto gera duas possíveis transições. A primeira verifica se ainda há itens a anunciar, caso em que o processo volta ao estado composto **Leiloando Item**, ou seja, trata-se de uma autotransição. Caso não haja mais itens a anunciar, gera-se uma transição para o último estado, no qual o leilão é encerrado, por meio do disparo do método `finalizaLeilao`, e o processo é concluído.

9.23.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

A solução para esse exercício está representada na figura 9.30, cujos estados serão explicados a seguir.

Esse processo se inicia quando o hóspede informa o número do quarto do qual deseja quitar as diárias, o que gera uma transição para o estado composto **Selecionando Diárias**. Nesse estado existem quatro subestados. O primeiro é responsável por consultar o quarto informado, onde será disparado o método `seDir`. Em seguida passa-se ao segundo subestado, onde é executado o método `con_Diarias_Aluga`, para fazer a ligação entre o quarto e o hóspede que o aluga. Após isso, passa-se ao terceiro subestado, responsável por consultar o hóspede que está alugando o quarto, no qual é executado o método `conHos`. Finalmente, passa-se para o último subestado, responsável por consultar as diárias devidas, disparando o método `con_Diaria`.

O término desse estado composto gera uma transição para o estado **Quitando Diárias**, no qual é executado o método `quit_Diaria`. A partir desse estado, uma transição encontra um pseudoestado de escolha, onde se deve testar se houve a solicitação de algum serviço no período em que o quarto estava ocupado, caso em que é gerada uma transição para um estado de submáquina referente ao processo de **Quitar Serviços**.

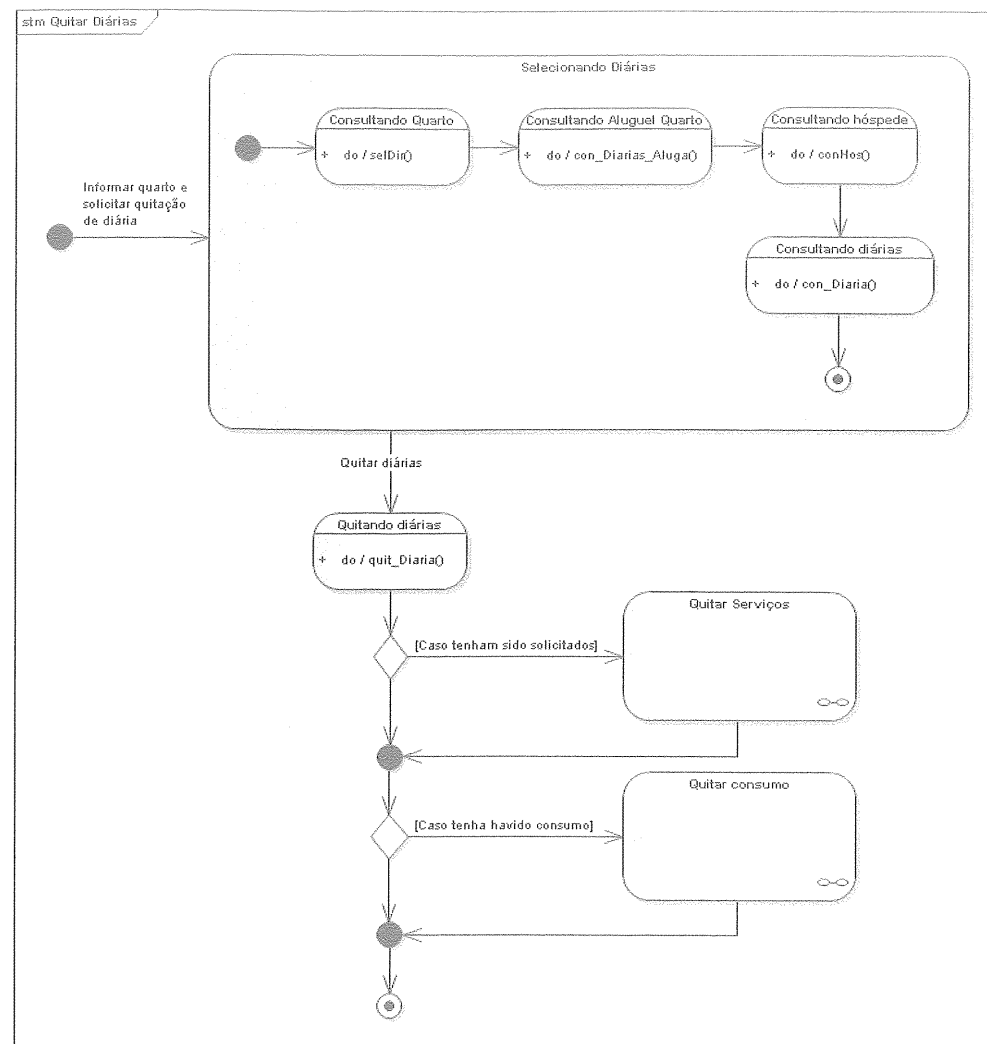


Figura 9.30 – Processo de Pagamento de Diárias.

O fluxo dividido pelo pseudoestado de escolha é reunido por um pseudoestado de junção que gera uma transição para um segundo pseudoestado de escolha, onde é necessário testar se houve algum consumo de frigobar, caso em que é gerada uma transição para um segundo estado de submáquina que representa o processo de **Quitar Consumo**. O fluxo dividido é reunido por um segundo pseudoestado de junção, e o processo é encerrado.

CAPÍTULO 10

Diagrama de Atividade

O diagrama de atividade era considerado um caso especial do antigo diagrama de gráfico de estados, chamado diagrama de máquina de estados atualmente. A partir da versão 2.0 da UML esse diagrama passou a ser considerado um diagrama totalmente independente, deixando inclusive de se basear em máquinas de estados e passando a se basear em redes de Petri.

A modelagem de atividade enfatiza a sequência e condições para coordenar comportamentos de baixo nível. Dessa forma, o diagrama de atividade é o diagrama com maior ênfase ao nível de algoritmo da UML e provavelmente um dos mais detalhistas. Esse diagrama apresenta muitas semelhanças com os antigos fluxogramas utilizados para desenvolver a lógica de programação e determinar o fluxo de controle de um algoritmo.

Este diagrama é utilizado, como o próprio nome diz, para modelar atividades, que podem ser um método ou um algoritmo, ou mesmo um processo completo. Atividades podem descrever computação procedural. Nesse contexto, elas são os métodos correspondentes às operações sobre classes. Atividades também podem ser aplicadas à modelagem organizacional para engenharia de processos de negócios e workflow. Finalmente, atividades podem também ser usadas para modelagem de sistemas de informação para especificar processos ao nível de sistema. Uma atividade é composta por um conjunto de ações, ou seja, os passos necessários para que a atividade seja concluída.

Um diagrama de atividade pode modelar mais de uma atividade. Uma atividade sempre conterá ações, no entanto, não necessariamente essas estarão representadas dentro da atividade, como quando for necessário fazer referência a uma atividade já modelada ou para poupar espaço no diagrama. Além disso, o diagrama de atividade pode ser usado para representar dois tipos de fluxo: de controle e de objetos, conforme será visto a seguir.

10.1 Atividade

Uma atividade especifica a coordenação de execuções de comportamentos subordinados usando um modelo de fluxo de controle e dados. Esses comportamentos subordinados podem ser iniciados devido a outros comportamentos no modelo terminarem sua execução; devido a objetos e dados tornarem-se disponíveis; ou pela ocorrência de eventos externos. Uma atividade é representada por um retângulo grande com as bordas arredondadas, conforme demonstra a figura 10.1.

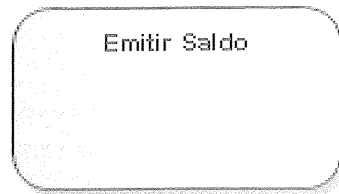


Figura 10.1 – Exemplo de Atividade.

Esse exemplo representa a atividade referente ao processo de emissão de saldo de uma conta, pertencente ao sistema de controle bancário que estamos modelando ao longo do livro. Durante este capítulo detalharemos as etapas dessa atividade.

Atividades podem conter ações de vários tipos, tais como:

- ocorrências de funções primitivas, tais como funções aritméticas;
- invocação de comportamento, tais como atividades;
- ações de comunicação, tais como envio de sinais;
- manipulação de objetos, tais como leitura ou gravação de atributos ou mesmo instanciação ou destruição de objetos.

10.2 Nó de Ação

Nós de ação são os elementos mais básicos de uma atividade. Um nó de ação representa um passo, uma etapa que deve ser executada em uma atividade. Um nó de ação é atômico, não podendo ser decomposto. Um nó de ação é representado por um pequeno retângulo com as bordas arredondadas, semelhante a uma atividade, porém o símbolo do nó de ação é menor. A figura 10.2 apresenta um exemplo de nó de ação.



Figura 10.2 – Nó de Ação.

Esse exemplo representa a ação inicial da atividade de emissão de saldo, onde se deve receber o número da conta informada pelo cliente.

10.3 Fluxo de Controle

O fluxo de controle é um conector que liga dois nós, enviando sinais de controle. É representado por uma linha contendo uma seta apontando para o novo nó e partindo do antigo, podendo conter uma descrição, uma condição de guarda ou uma restrição, chamada nesse diagrama de peso (weight), que determina, por exemplo, o número mínimo de sinais que devem ser transmitidos pelo fluxo. Um sinal (token) pode conter valores de controle, objetos ou dados, sendo que esses dois últimos somente podem ser transmitidos por meio de um fluxo de objetos, que será explicado neste capítulo. A figura 10.3 apresenta um exemplo de fluxo de controle.



Figura 10.3 – Fluxo de Controle.

Nesse exemplo, que dá continuidade ao anterior, há dois nós de ação. A primeira ação representa o recebimento do número de uma conta, enquanto a segunda representa a consulta da conta informada. Observe que a passagem de uma ação para outra é representada pela linha do fluxo de controle que une as duas ações. Note ainda que a seta do fluxo de controle demonstra a ordem em que as ações serão executadas.

10.4 Nó Inicial

Esse componente pertence ao grupo de nós de controle utilizados para o controle de fluxo da atividade. Esse nó é usado para representar o início do fluxo quando a atividade é invocada. É representado por um círculo preenchido. A figura 10.4 apresenta um exemplo de nó inicial.

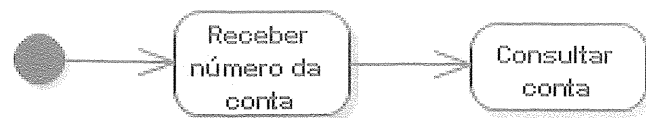


Figura 10.4 – Nó Inicial.

Como o leitor pode observar, complementamos o exemplo anterior acrescentando um nó inicial à figura, indicando o início do diagrama.

10.5 Nó de Final de Atividade

Esse componente é também um nó de controle usado para representar o fim do fluxo de uma atividade. É representado por um círculo preenchido dentro de um círculo vazio. A figura 10.5 apresenta um exemplo de nó final.



Figura 10.5 – Nó de Final de Atividade.

Aqui apresentamos o final do fluxo iniciado nas figuras anteriores (obviamente faltam ações intermediárias), onde, depois de apresentar o saldo da conta, o fluxo da atividade é encerrado.

10.6 Nó de Decisão

Um nó de decisão é também um nó de controle, utilizado para representar uma escolha entre dois ou mais fluxos possíveis, em que um dos fluxos será escolhido em detrimento dos outros. Em geral um nó de decisão é acompanhado por condições de guarda, ou seja, textos entre colchetes que determinam a condição

para que um fluxo possa ser escolhido. Um nó de decisão pode ser utilizado também para unir um fluxo dividido por um nó de decisão anterior, quando passa a chamar-se nó de união. A figura 10.6 apresenta um exemplo de nó de decisão.

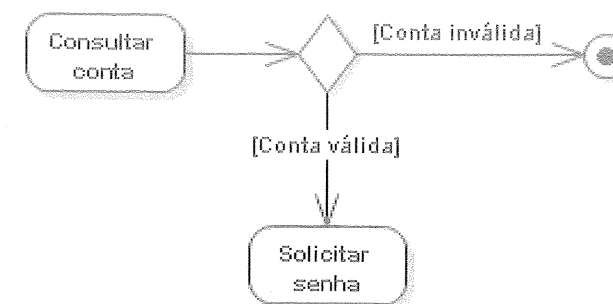


Figura 10.6 – Nó de Decisão.

Nesse exemplo, continuamos a atividade que representa o processo de emissão de saldo, onde, depois de consultar a conta, deve-se tomar uma decisão, representada por um losângulo. Caso a conta seja inválida, o processo deve ser encerrado, caso contrário, deve-se solicitar sua senha. Observe que as condições dessa decisão estão definidas por condições de guarda (texto entre colchetes), posicionados sobre os dois fluxos alternativos.

10.7 Exemplo Simples de Diagrama de Atividade

Nesta seção demonstraremos um exemplo de diagrama de atividade, onde modelamos a atividade completa de emissão de saldo, que já foi iniciada nas figuras anteriores. A atividade é apresentada pela figura 10.7.

Como essa atividade já foi parcialmente explicada nos exemplos anteriores, iniciaremos sua descrição a partir da ação seguinte ao nó de ação que solicita a senha da conta, que representa o recebimento da senha solicitada. Depois de a senha ter sido fornecida pelo cliente, esta deve ser validada, representada pelo nó de ação **Validar senha**, o que leva a um nó de decisão onde, se a senha for inválida, deve-se encerrar o fluxo da atividade, caso contrário, passa-se para a ação que consulta o saldo da conta. Quando essa ação é concluída, passa-se à ação que apresenta o saldo na interface e encerra-se a atividade.

Como já foi dito, esse é um exemplo simples de diagrama de atividade. A seguir demonstraremos outros componentes desse diagrama e apresentaremos exemplos mais complexos.

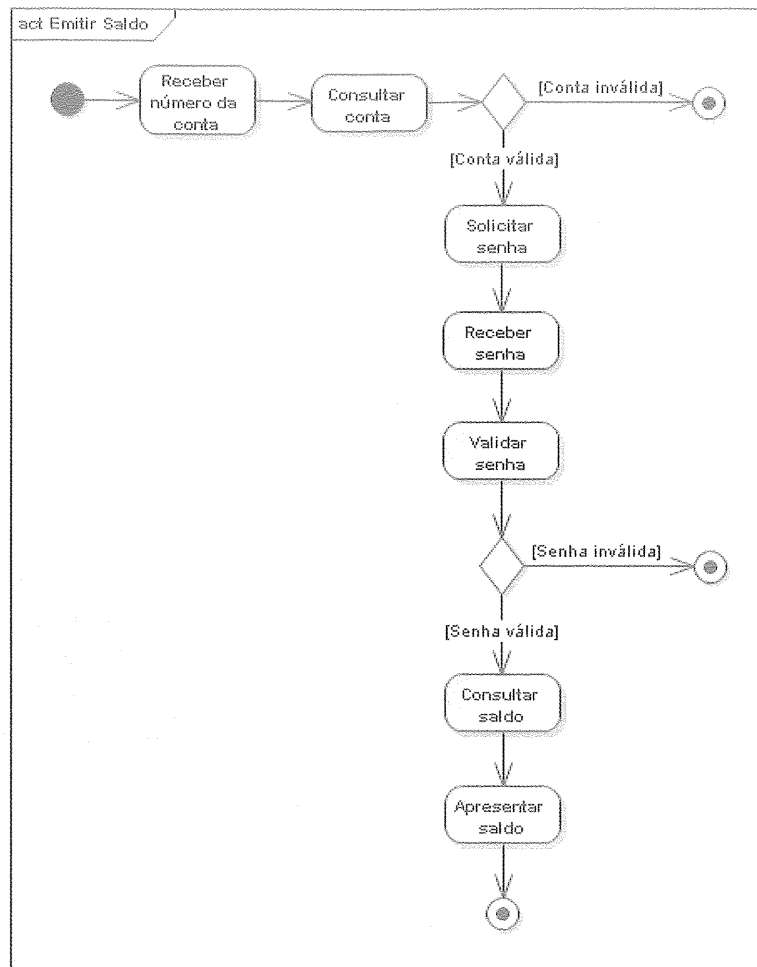


Figura 10.7 – Exemplo de Diagrama de Atividade – Processo de Emissão de Saldo.

10.8 Nó de Bifurcação/União

Um nó de bifurcação/união é um nó de controle que pode tanto dividir um fluxo em dois ou mais fluxos concorrentes, quando é chamado nó de bifurcação, como mesclar dois ou mais fluxos concorrentes em um único fluxo de controle, quando é chamado nó de união. Esse nó é representado por uma barra que pode estar tanto na horizontal como na vertical. A figura 10.8 apresenta um exemplo de nó de bifurcação/união.

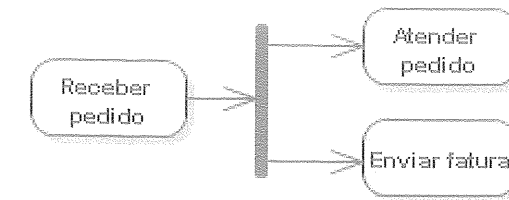


Figura 10.8 – Exemplo de Nó de Bifurcação/União.

Aqui apresentamos um exemplo onde, a partir do nó de ação *Receber Pedido*, o fluxo de controle é dividido em dois fluxos paralelos, como demonstra o nó de bifurcação/união. A partir desse momento os nós de ação *Atender pedido* e *Enviar fatura* serão executados paralelamente.

10.9 Final de Fluxo

Representa o encerramento de uma rotina representada pelo fluxo, mas não de toda a atividade. O símbolo de final de fluxo é representado por um círculo com um X, conforme demonstrado na figura 10.9, onde assumimos uma situação em que muitos componentes podem ser construídos e instalados. Quando o último componente for construído, essa parte do fluxo estará concluída, como demonstra o símbolo de final de fluxo. No entanto, outras etapas deverão ser concluídas durante a atividade.

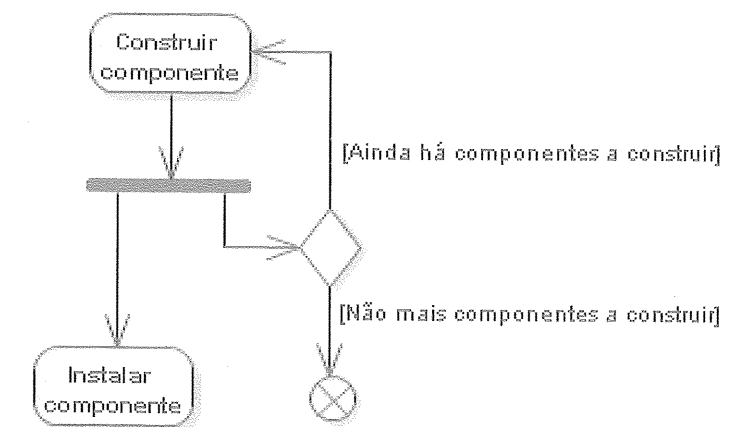


Figura 10.9 – Exemplo de Final de Fluxo.

10.10 Fluxo de Objetos

Um fluxo de objetos é um conector que pode ter objetos ou dados passando por ele. Representa o fluxo de valores (objetos ou dados) que são enviados a partir de um nó de objeto ou para um nó de objeto. Na seção seguinte apresentaremos um exemplo de fluxo de objetos.

10.11 Nó de Objeto

Um nó de objeto representa uma instância de uma classe, que pode estar disponível em um determinado ponto da atividade. Nós de objeto podem ser utilizados de diversas formas. Em sua forma mais tradicional, são representados como um retângulo, como demonstra a figura 10.10.

O fluxo de objetos pode ser utilizado para modificar o estado de um objeto, definindo um valor para um de seus atributos ou mesmo instanciando ou destruindo o objeto. Um objeto pode apresentar multiplicidade, que nesse caso determina o número mínimo e máximo de valores que o objeto aceita. Se existir um valor mínimo determinado, a ação só inicia quando ele for atingido. Um objeto pode apresentar também um “Upperbound” entre colchetes que determina o valor máximo de valores que o objeto pode conter. Em tempo de execução, quando esse valor é ultrapassado, o fluxo é interrompido. A figura 10.10 apresenta um exemplo de nó de objeto junto com fluxo de objetos, que é muito semelhante ao fluxo de controle.

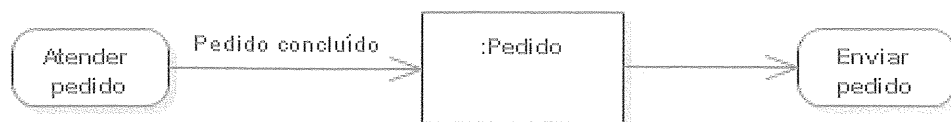


Figura 10.10 – Exemplo de Nó de Objeto e Fluxo de Objeto.

Neste exemplo, após o atendimento do pedido ter sido concluído, atualiza-se um objeto da classe Pedido para determinar que este foi concluído, passando-se em seguida para o nó de ação Enviar pedido.

10.12 Alfinetes (Pins)

O fluxo de objetos muitas vezes exige a utilização de alfinetes (pins). Alfinetes são nós de objeto que representam uma entrada para uma ação ou uma saída de uma ação. Eles fornecem valores para as ações e recebem os valores resultantes delas. O próprio objeto apresentado na figura anterior é um alfinete. Os

alfinetes podem ser representados também como pequenos retângulos e em geral são colocados ao lado das ações às quais estão ligados, conforme demonstra a figura 10.11. Quando o tipo de entrada e saída é o mesmo, pode-se usar um único retângulo no centro do fluxo de dois nós de ação, conforme apresentado anteriormente na figura 10.10.



Figura 10.11 – Exemplo de Alfinetes.

Aqui apenas modificamos o exemplo da figura anterior, definindo que o nó de objeto pedido é uma informação de saída do nó de ação da esquerda e uma informação de entrada para o nó de ação da direita.

10.13 Nó de Parâmetro de Atividade

Um nó de parâmetro de atividade é um nó de objeto utilizado para representar a entrada ou a saída de um fluxo de objetos em uma atividade. A figura 10.12 apresenta um exemplo de nó de parâmetro de atividade.

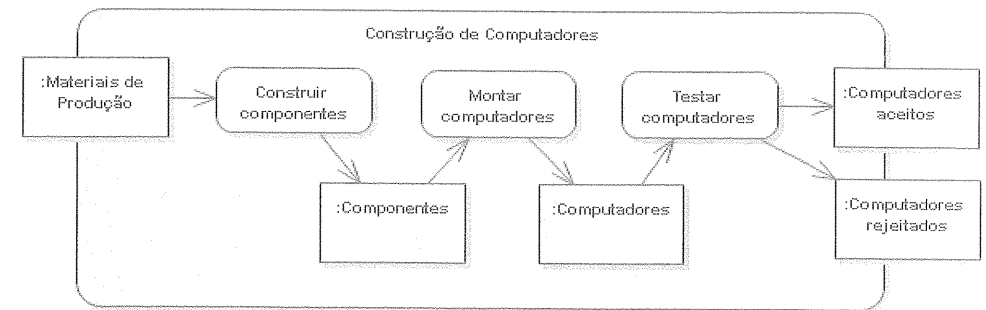


Figura 10.12 – Exemplo de Parâmetro de Atividade.

Nessa figura, o nó de objeto Materiais de Produção é um nó de parâmetro de entrada na atividade, enquanto os nós de objeto Computadores Aceitos e Computadores Rejeitados são nós de parâmetro de saída na atividade. Observe que a atividade contém, em seu interior, três nós de ação, denominados Construir componentes, Montar computadores e Testar computadores, e que esses nós de ação enviam e recebem informações para os nós de objeto (que são alfinetes) Componentes e Computadores.

10.14 Exceções

É possível, no diagrama de atividade, detalhar a ocorrência de exceções, bastante comuns na maioria das linguagens de programação atuais. Para descrever a manipulação de uma exceção, o diagrama de atividade fornece uma seta em forma de raio que aponta para a rotina de tratamento da interrupção ou exceção. Essa seta é chamada de fluxo de interrupção, conforme é demonstrado na figura 10.13.

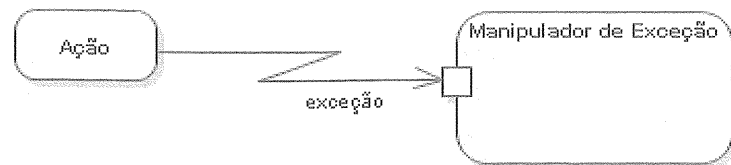


Figura 10.13 – Tratamento de Exceção.

Observe que a seta de exceção atinge um quadrado no manipulador de exceção. Esse quadrado é também um nó de objeto, chamado nó de entrada de exceção.

10.15 Ação de Envio de Sinal (Ação de Objeto de Envio na versão 2.0 anterior)

É uma ação que representa o envio de um sinal para um objeto ou ação. Uma ação de envio de sinal é representada por um retângulo com uma protuberância triangular em seu lado direito. Na próxima seção apresentaremos um exemplo de ação de envio de sinal.

10.16 Ação de Evento de Aceitação

É uma ação que representa a espera de ocorrência de um evento de acordo com determinadas condições. Uma ação de evento de aceitação é representada por um retângulo com uma reentrância triangular em seu lado esquerdo. A figura 10.14 apresenta um exemplo de ações de envio de sinal e de evento de aceitação.

Nesse exemplo, utilizamos um fluxo de controle referente ao envio de um texto para impressão. Depois de preparar o texto, é enviado um sinal à impressora para verificar se esta está pronta para receber o material a ser impresso, em seguida a impressora envia um sinal em resposta, informando que ela pode imprimir o documento. Observe que a impressora é representada como um objeto, e que os sinais são enviados e recebidos por meio de um fluxo de objeto.

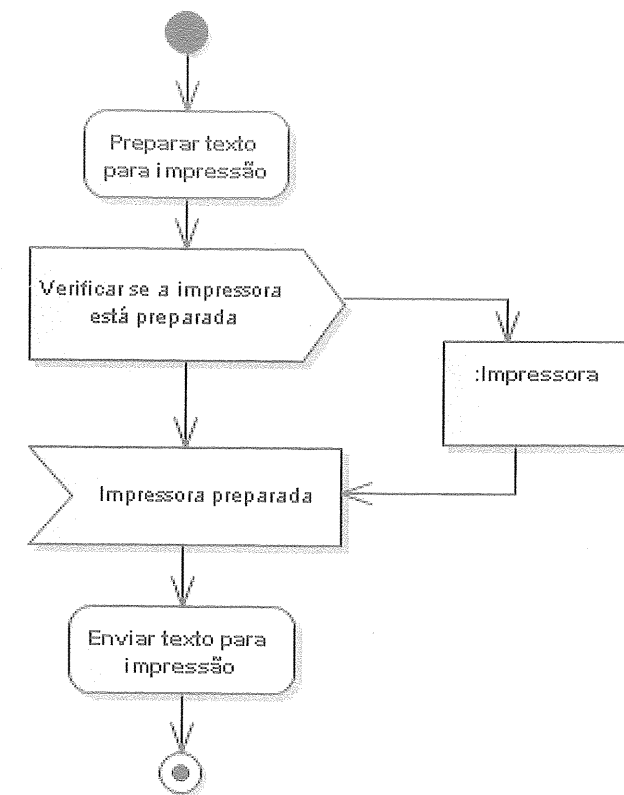


Figura 10.14 – Ações de Envio de Sinal e de Evento de Aceitação.

10.17 Ação de Evento de Tempo de Aceitação

É uma variação da ação de evento de aceitação que leva em consideração o tempo para que o evento possa ser disparado. Pode ser comparada com um trigger. A figura 10.15 apresenta um exemplo de ação de evento de tempo de aceitação.

Nesse exemplo, quando o horário de final de expediente for atingido, é disparada a atividade denominada **Realizar backup automático**.

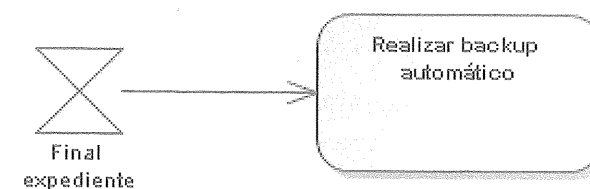


Figura 10.15 – Ação de Evento de Tempo de Aceitação.

10.18 Nó de Buffer Central

Um nó de buffer central é um nó de objeto cuja função é gerenciar fluxos de múltiplas fontes e destinos. Ele age como um buffer de memória para múltiplos fluxos de entrada e fluxos de saída de outros nós de objeto. Ele não se conecta diretamente a ações. É representado como um nó de objeto com o estereótipo <<centralBuffer>>.

10.19 Nó de Repositório de Dados (Data Store Node)

Esse nó é um tipo especial de nó de buffer central. Um nó de repositório de dados é um nó de objeto representado com o estereótipo de <<datastore>>. Um nó de repositório de dados é usado para armazenar dados ou objetos permanentemente. Os dados que entram em um nó de repositório de dados são armazenados permanentemente, atualizando os dados que já existiam. Os dados que vem de um nó de repositório de dados são uma cópia dos dados originais. A figura 10.16 apresenta um exemplo de nó de repositório de dados.

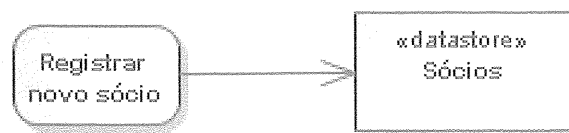


Figura 10.16 – Nó de Repositório de Dados.

Nesse exemplo estamos demonstrando que a ação Registrar novo sócio envia informações por meio de um fluxo de objetos para um repositório que armazenará essas informações de forma permanente.

10.20 Conectores

Conectores são basicamente atalhos para o fluxo, utilizados quando existe uma distância relativamente grande entre os nós que o fluxo precisa ligar. Um conector é representado por um círculo contendo uma letra, por exemplo. Deve haver sempre pares de conectores com a mesma nomenclatura, uma vez que um conector é um atalho. Assim, se houver um fluxo de entrada para o conector A, deve haver em outra parte do diagrama um fluxo de saída de um conector de mesmo nome. A figura 10.17 apresenta um exemplo simples de conectores. Outros exemplos poderão ser vistos no estudo de caso sobre o sistema de pizzaria on-line, no capítulo 16.



Figura 10.17 – Exemplo de Conectores.

10.21 Ação de Chamada de Comportamento

Uma ação de chamada de comportamento é uma ação que invoca a execução de um comportamento, sendo este, em geral, uma atividade. Esse tipo de ação apresenta um símbolo de ancinho apontando para baixo em seu canto inferior direito. A figura 10.18 apresenta um exemplo desse tipo de ação.

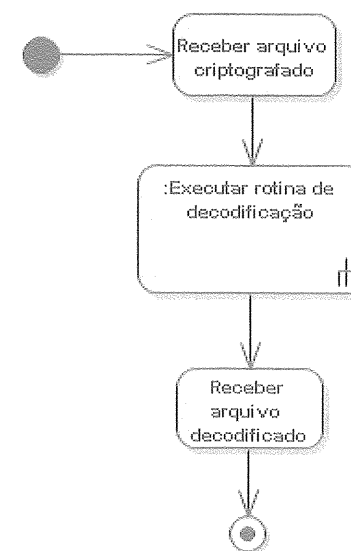


Figura 10.18 – Exemplo de Ação de Chamada de Comportamento.

A figura 10.18 ilustra um exemplo em que um arquivo criptografado é recebido e repassado para uma rotina de decodificação. Como as etapas para decodificar o arquivo são desconhecidas ou já foram modeladas em outro diagrama, não há necessidade de defini-las novamente, por isso utilizamos uma ação de chamada de comportamento para invocar esta atividade.

10.22 Ação de Chamada de Operação

Enquanto a ação de chamada de comportamento é uma chamada direta a um comportamento, uma ação de chamada de operação é uma chamada indireta a um comportamento, na qual é invocada a execução de uma operação (método) em um objeto de uma classe. A figura 10.19 apresenta um exemplo desse tipo de ação.

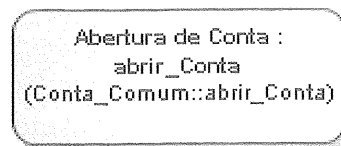


Figura 10.19 – Exemplo de Ação de Chamada de Operação.

Nesse exemplo uma ação de chamada de operação nomeada *Abertura de Conta* solicita a execução do método *abrir_Conta*. Embaixo da solicitação, entre parênteses, é mostrado a classe que possui a operação em questão, seguida de dois símbolos de dois pontos (:) e o nome da operação. O uso de uma ação de chamada de operação evita ter que descrever passo a passo todas as etapas da execução de um método em uma atividade.

10.23 Partição de Atividade

As partições de atividade permitem representar o fluxo de um processo que passa por diversos setores ou departamentos de uma empresa, ou mesmo um processo que é manipulado por diversos atores. As partições de atividade são formadas por retângulos representando divisões que identificam as zonas de influência de um determinado setor sobre parte de um determinado processo. As partições podem ser tanto horizontais como verticais. A figura 10.20 apresenta um exemplo de partições de atividade.

Nesse exemplo existem três partições de atividade, representando o setor de vendas, o setor de contabilidade e o cliente, representando as pessoas que compram produtos da empresa. O início do processo é identificado pela ação que demonstra o recebimento de um pedido no setor de vendas. A ação seguinte representa o atendimento desse pedido e, quando esta for concluída, passa-se para um nó de bifurcação que divide o fluxo em dois, mantendo um fluxo no setor de vendas e passando o segundo fluxo para o setor de contabilidade. Assim, ao mesmo tempo em que o pedido é enviado pelo setor de vendas, a fatura é enviada pelo setor de contabilidade.

O envio da fatura gera um objeto da classe *Fatura* que armazenará as informações da fatura gerada. O conteúdo desse objeto será enviado ao cliente, que realizará a ação de *Realizar pagamento*. O recebimento do pagamento pelo setor de contabilidade reunirá novamente o fluxo dividido anteriormente no setor de vendas, como demonstra o nó de união, produzindo a ação *Fechar pedido* e concluindo o processo.

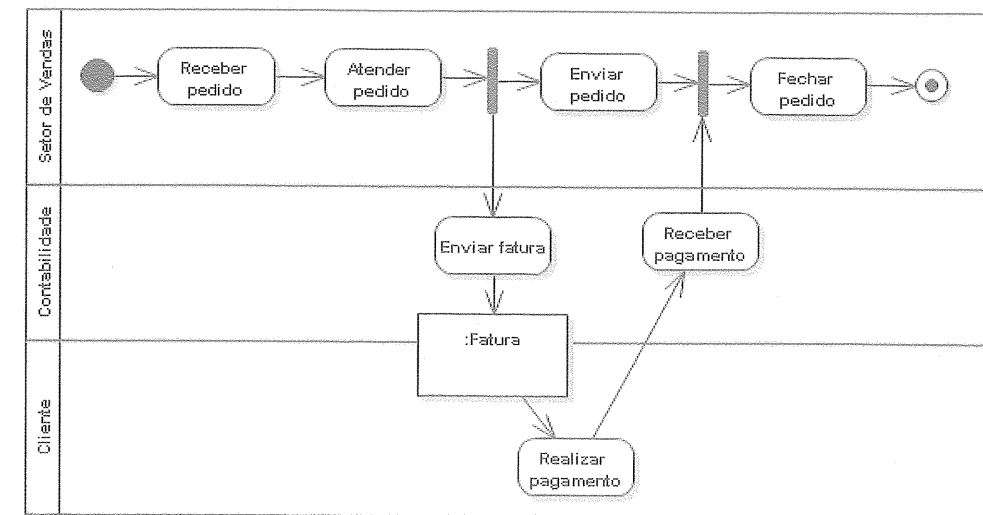


Figura 10.20 – Partições de Atividade.

10.24 Região de Atividade Interrompível

Uma região de atividade interrompível engloba certo número de elementos da atividade que podem sofrer uma interrupção, assim, todo o processo envolvido pela região poderá ser interrompido, não importando em que elemento da atividade a interrupção ocorra. A região é delimitada por um retângulo tracejado com as bordas arredondadas. É necessário utilizar uma ação de envio de sinal para indicar a interrupção, além do uso de um fluxo de interrupção, conforme demonstra o exemplo da figura 10.21, onde a atividade representada na figura anterior foi melhorada para permitir a opção de cancelamento do pedido.

Nesse exemplo, podemos perceber que a região envolvendo os nós de ação *Receber pedido*, *Atender pedido* e *Enviar pedido* é uma região de atividade interrompível, ou seja, o pedido pode ser cancelado. Porém, a partir do momento que o pagamento for recebido, não é mais possível cancelar o pedido. Observe que uma ação de envio de sinal gera um fluxo de exceção para o nó de ação *Cancelar pedido*, que encerra o processo, caso o cliente opte por essa alternativa.

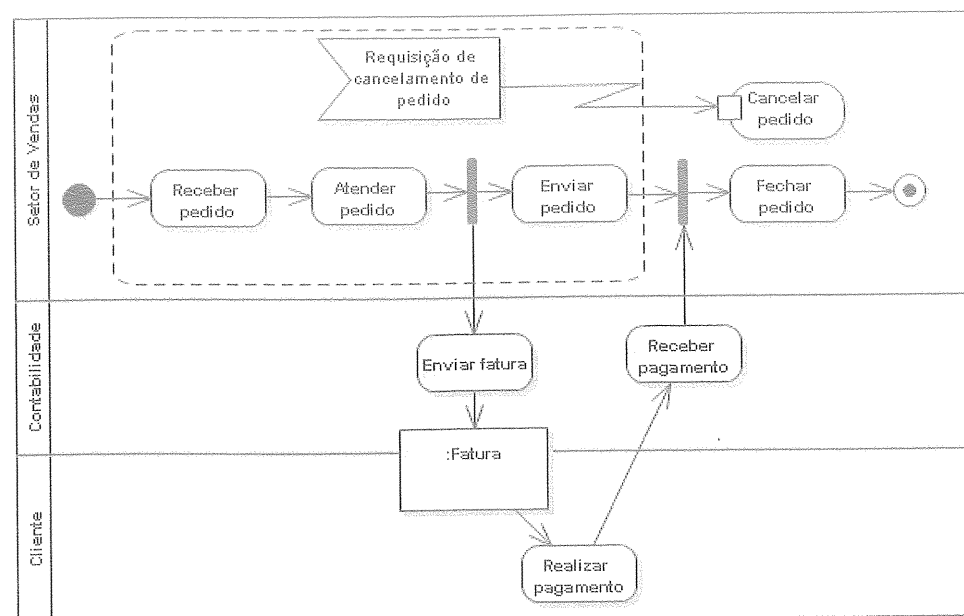


Figura 10.21 – Exemplo de Região de Atividade Interrompível.

10.25 Nó de Atividade Estruturada

Um nó de atividade estruturada é um nó de atividade executável que pode conter nós subordinados. Esses nós subordinados devem pertencer somente a um nó de atividade estruturada. Um nó de atividade estruturada representa uma porção estruturada da atividade que não é compartilhada com qualquer outro nó estruturado, exceto para aninhamento.

Devido à natureza concorrente da execução de ações dentro e ao longo das atividades, pode ser difícil garantir o acesso e modificação consistente da memória do objeto. Dessa forma, às vezes é necessário isolar os efeitos de um grupo de ações dos efeitos de ações fora do grupo.

Um nó de atividade estruturada é representado como um símbolo de atividade, mas com suas bordas tracejadas e contendo um símbolo de diagrama de atividade em seu canto inferior direito, o que significa que as ações desta atividade podem estar detalhadas em outro diagrama. Além disso, esse nó contém o estereótipo <<structure>>. A figura 10.22 apresenta um exemplo desse nó, onde adaptamos o exemplo demonstrado na seção 10.20, substituindo a ação de chamada de comportamento por um nó de atividade estruturada.

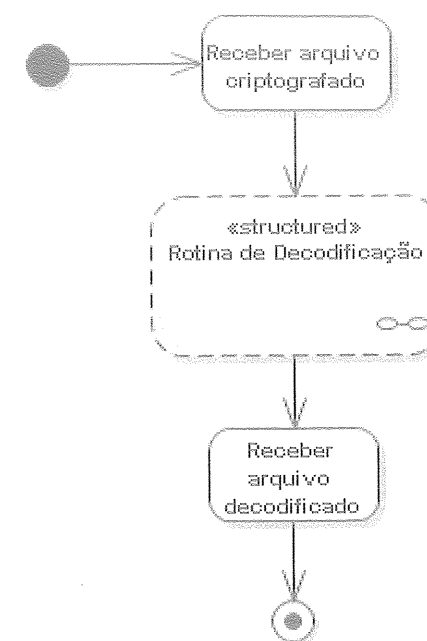


Figura 10.22 – Exemplo de Nó de Atividade Estruturada.

10.26 Região de Expansão

Uma região de expansão é uma região de atividade estruturada que é executada múltiplas vezes de acordo com os elementos de uma coleção de entrada. Uma região de expansão engloba uma parte da atividade e representa uma região aninhada na qual cada entrada é uma coleção de valores. A região de expansão é executada uma vez para cada elemento na coleção de entrada. A cada execução da região, um valor de saída é inserido na coleção de saída, na mesma posição que os elementos de entrada. Os elementos de entrada e saída são nós de objeto, cada um representado por três pequenos quadrados juntos, colocados nas extremidades da região de expansão, aqui chamados de nós de expansão.

Uma região de expansão pode ser iterativa, onde as interações ocorrem na ordem dos elementos; paralela, onde todas as interações são independentes; ou de fluxo (stream), onde há uma única execução da região em que os valores na coleção de entrada são extraídos e colocados na execução da região de expansão como um fluxo. A figura 10.23 apresenta um exemplo simples de região de expansão, onde foi modelado um cálculo recursivo de fatorial, que deverá ser chamado várias vezes.

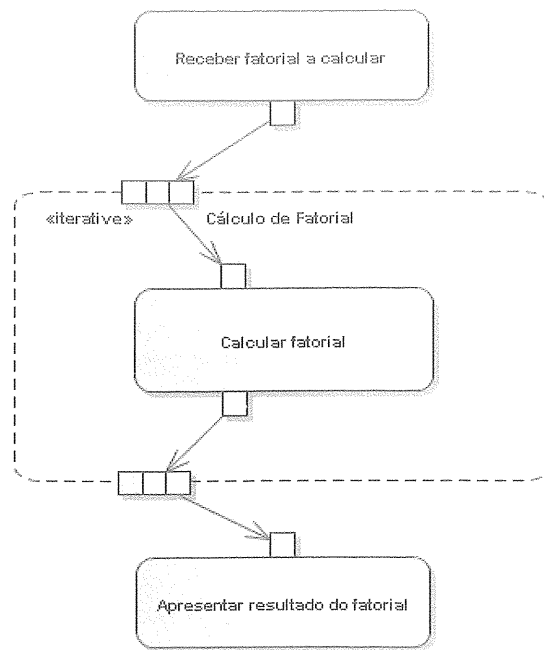


Figura 10.23 – Exemplo de Região de Expansão.

10.27 Exemplo de Diagrama de Atividade – Emitir Extrato

Nesta e nas seções seguintes apresentaremos os processos referentes ao sistema de controle bancário que estamos modelando ao longo deste livro. Aqui apresentamos o diagrama de atividade para o processo de emissão de extrato, ilustrado pela figura 10.24, que será explicado nesta seção.

A primeira ação desse processo é esperar que o cliente informe o número da conta da qual deseja emitir o extrato. Depois de receber o número da conta, passa-se a ação de consultá-la e, em seguida, a um nó de decisão, no qual é verificado se a conta é inválida, caso em que o processo é encerrado, ou passa-se à ação de solicitar senha, caso a conta seja válida. Na ação seguinte o processo fica aguardando que a senha da conta seja informada e, depois de ser fornecida, passa-se à ação de validação de senha, que passa a um nó de decisão cuja função é determinar se o processo deve ser encerrado, caso a senha seja inválida, ou se é preciso passar à ação de solicitação dos períodos desejados para o extrato.

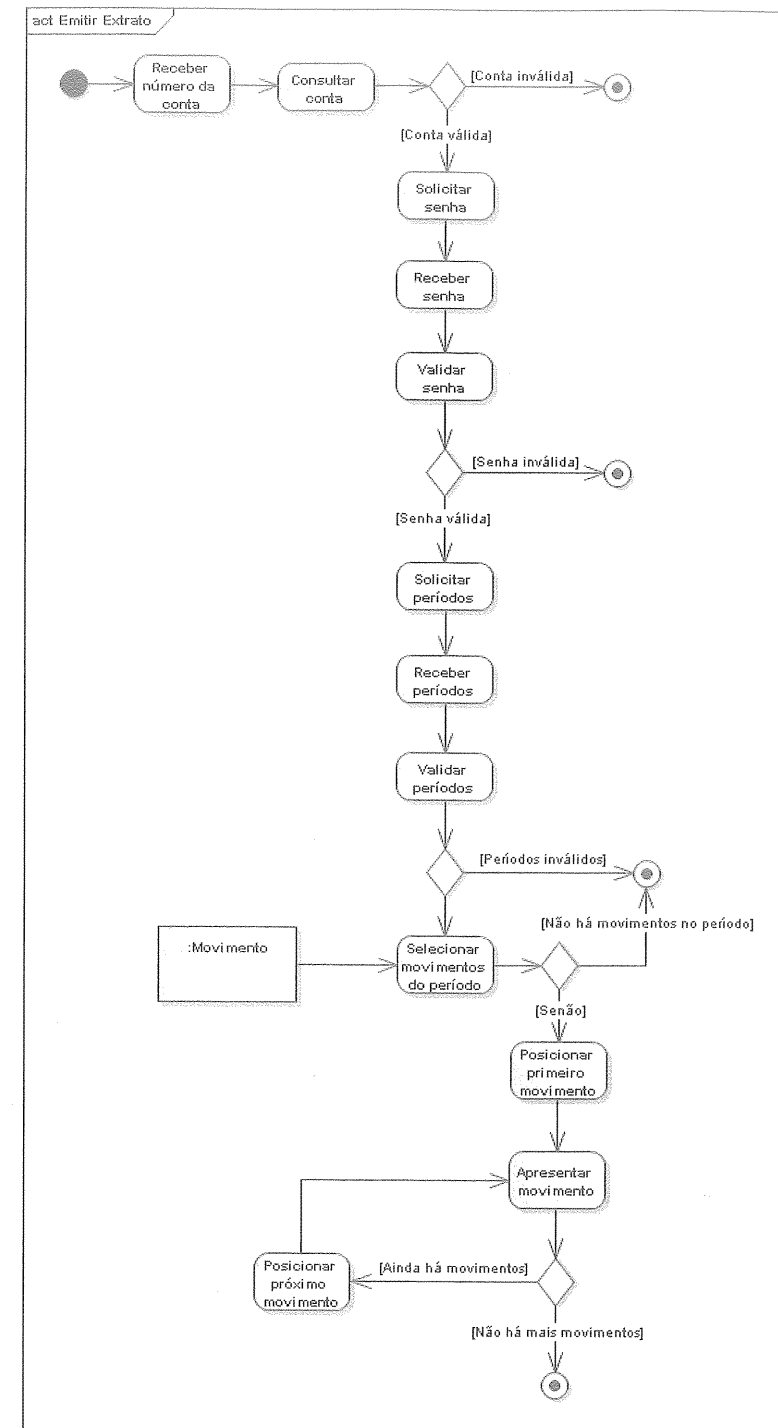


Figura 10.24 – Processo de Emissão de Extrato.

Depois de receber os períodos, passa-se à ação de validação dos períodos para determinar se estes são períodos corretos (o período inicial não pode ser maior que o final). Um nó de decisão determina se o processo deve ser encerrado, caso os períodos sejam inválidos, ou se é preciso passar ao nó de ação **Selecionar movimentos do período**. Observe que essa última ação tem um fluxo de objetos com um objeto da classe **Movimento**, significando que para selecionar os movimentos do período é necessário consultar objetos dessa classe. A rigor deveríamos ter feito o mesmo quando consultamos a conta ou validamos a senha, mas achamos desnecessário esse nível de detalhe.

A partir dessa ação passa-se a um novo nó de decisão, que determina se o processo deve ser encerrado, caso não tenham sido encontrados movimentos no período, ou se é necessário passar à ação **Posicionar primeiro movimento**, onde o processo escolhe o primeiro movimento encontrado na seleção anterior. Após isso, o movimento é apresentado na ação seguinte e passa-se a um novo nó de decisão, cuja função é determinar se ainda há movimentos, caso em que passa-se à ação de posicionar no objeto **Movimento** seguinte e volta-se à ação de apresentar o movimento. Caso não haja mais movimentos, deve-se encerrar o processo.

10.28 Exemplo de Diagrama de Atividade – Realizar Depósito

Nesta seção enfocamos o processo de **Realizar Depósito** do sistema de controle bancário, conforme representado na figura 10.25.

As primeiras ações desse processo são idênticas ao anterior, passando a ser diferentes a partir da ação que solicita o valor para o depósito. A ação seguinte recebe o valor solicitado e passa à ação de somar o valor ao saldo da conta. Observe que há um fluxo de objetos para um objeto da classe **Conta_Comum** (herdado por objetos de suas subclasses) que recebe o valor transmitido por esse fluxo. Algo semelhante ocorre na ação seguinte, que registra o movimento e tem um fluxo de objetos para um objeto da classe **Movimento**, representando sua instanciiação. Após essa ação, o processo é finalizado.

Poderíamos ter colocado a última ação em um diagrama separado, mas por esta ser apenas uma preferimos inseri-la junto a este diagrama.

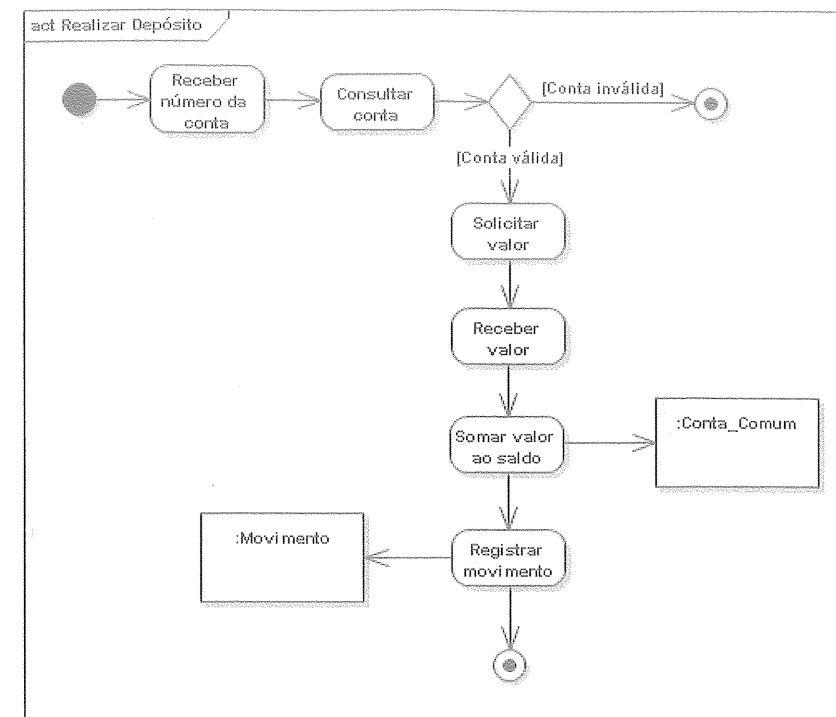


Figura 10.25 – Processo de Realizar Depósito.

10.29 Exemplo de Diagrama de Atividade – Realizar Saque

Passamos aqui ao processo de **Realizar Saque** do sistema de controle bancário, conforme pode ser visto na figura 10.26.

As primeiras ações desse processo são idênticas às do processo de **Emitir Extrato** até o momento em que a senha é validada, e depois disso as ações são muito semelhantes ao processo de **Realizar Depósito**, exceto pela ação que diminui o valor do saldo da conta, a única diferente. O fluxo de objetos também representa ações semelhantes.

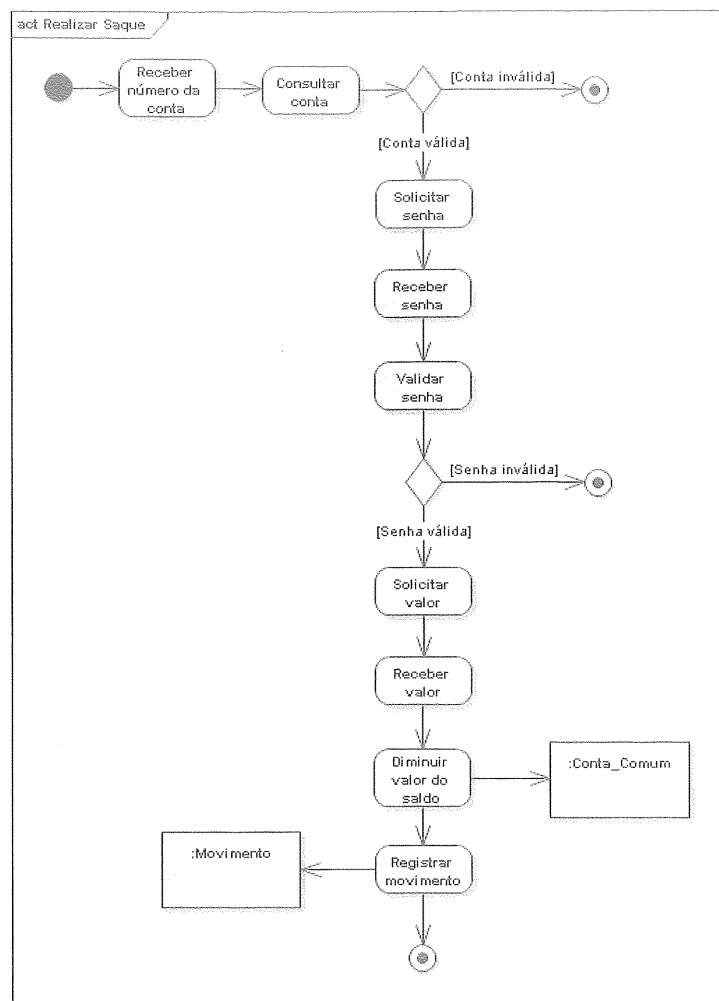


Figura 10.26 – Processo de Realizar Saque.

10.30 Exemplo de Diagrama de Atividade – Abrir Conta Comum

Esse processo é um pouco mais complexo, como podemos observar pela figura 10.27. A seguir iremos explicar suas atividades e ações.

O leitor notará que este processo tem três atividades, sendo que a principal, que se refere ao processo de abertura de conta propriamente dito, detalha seus nós de ação, enquanto as atividades de *Manter Cliente* e *Realizar Depósito* não o fazem, sendo apenas referenciadas. O leitor poderá observar também que a comunicação entre essas atividades é feita por meio de nós de parâmetro de atividade. A seguir detalharemos esse diagrama, iniciando pela atividade de abertura de conta comum.

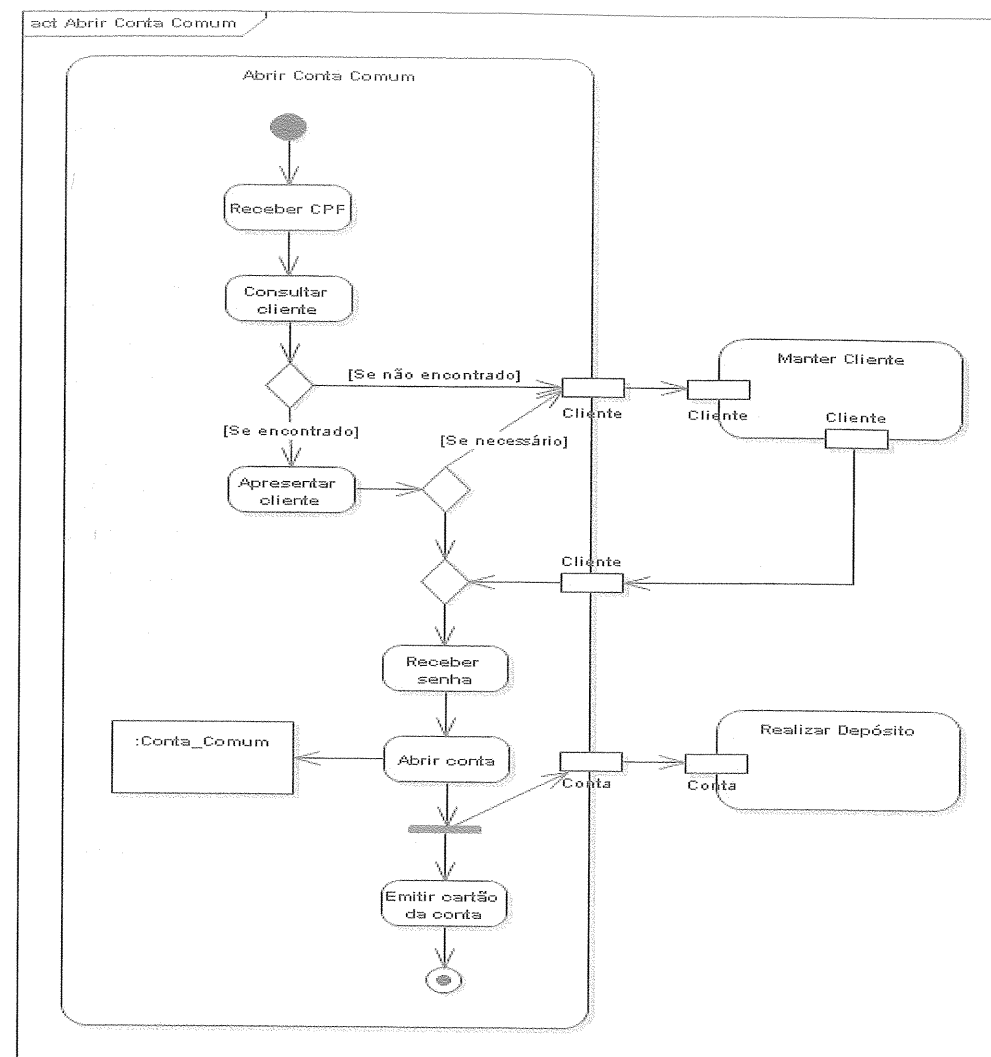


Figura 10.27 – Processo de Abertura de Conta Comum.

A atividade inicia-se com o processo recebendo o CPF do cliente que deseja abrir uma conta. A ação seguinte consulta o CPF informado e passa a um nó de decisão, que determina que o cliente deve ser registrado, se ele não for encontrado, ou passar à ação que apresenta os dados do cliente, que passa em seguida a um novo nó de decisão, cuja função é estabelecer se é preciso atualizar os dados do cliente ou passar à ação que recebe a senha da nova conta a ser aberta.

Observe que nas duas situações em que se deve registrar ou alterar o cadastro do cliente a comunicação é feita por meio de nós de parâmetro de atividade, onde primeiramente é comunicada a informação do cliente para a atividade de *Manter Cliente*, na qual este é cadastrado ou alterado. Em seguida as informações

do cliente são devolvidas atualizadas ao processo de abertura de conta por meio de outros nós de parâmetro atividade, mas que desempenham a mesma função e recebem o mesmo tipo de objeto. Note ainda que o fluxo dividido pelo segundo nó de decisão é unido por um nó de união, que apresenta o mesmo símbolo do de decisão. Utilizamos nós de parâmetro de atividade porque há uma suspensão temporária da atividade de abertura de conta para se executar outra atividade, e esta precisa de parâmetros iniciais para executar corretamente.

Depois de ter seu fluxo unido, a atividade de abertura de conta comum recebe a senha da nova conta a ser aberta e passa à sua abertura propriamente dita. Nessa ação há um fluxo de objetos que representa a instanciação de um novo objeto da classe *Conta_Comum*. A seguir, a atividade passa a um nó de bifurcação que divide o fluxo em dois, onde o primeiro solicita a execução da atividade *Realizar Depósito* e o segundo emite o cartão da nova conta gerada, depois do que a atividade é encerrada. Observe que novamente usamos nós de parâmetro de atividade para enviar a conta gerada para a atividade de *Realizar Depósito*.

10.31 Exemplo de Diagrama de Atividade – Encerrar Conta

O diagrama aqui demonstrado refere-se ao processo de encerramento de conta, ilustrado pela figura 10.28. Explicaremos esse diagrama a seguir.

O leitor perceberá que nesse processo utilizamos uma série de atividades que se referem a processos já modelados previamente, por este motivo não detalhamos suas ações internas, o que permite deixar o diagrama menor e mais fácil de compreender.

A primeira atividade a ser executada é a atividade de *Emitir Saldo*, uma vez que é necessário saber o saldo da conta antes de encerrá-la. Em seguida é feito um teste: se o saldo da conta for positivo, é executada a atividade *Realizar Saque*, caso contrário é executada a atividade *Realizar Depósito*. Observe que o fluxo dividido é unido por um nó de união.

Em seguida o fluxo cai novamente em um nó de decisão que verifica se será necessário ou não efetuar manutenção no registro do cliente, se a conta a ser encerrado for a única por ele possuída. Em caso positivo é executada a atividade *Manter Cliente*. Os dois fluxos divididos são unidos por um novo nó de união.

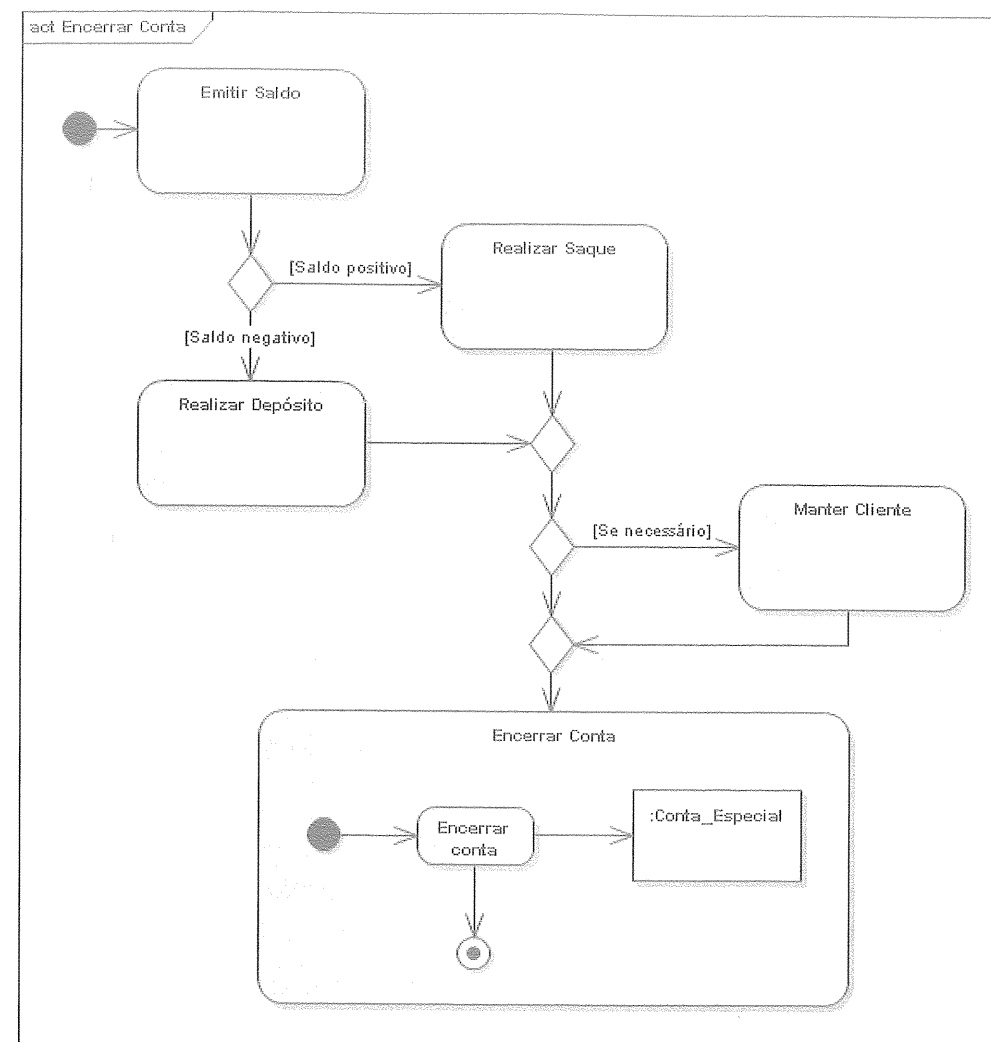


Figura 10.28 – Processo de Encerramento de Conta.

Finalmente, passa-se à atividade principal do diagrama, que representa o encerramento da conta propriamente dita, cujos nós de ação se resumem apenas à ação de encerrar conta. Observe que há um fluxo de objetos entre essa ação e um objeto da classe *Conta_Especial*, que representa a alteração do atributo *situacao* desse objeto para inativo. Após a conclusão da ação, o processo é encerrado.

10.32 Exercícios Propostos

Como já tem ocorrido nos capítulos anteriores, continuaremos a modelagem dos sistemas já iniciados, enfocando, desta vez, o diagrama de atividade.

10.32.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Desenvolva o diagrama de atividade referente ao processo de venda de ingressos para um sistema de controle de cinema sabendo que:

- Ao selecionar a opção de venda de ingressos, o sistema deverá apresentar todas as sessões ainda não encerradas. Cada sessão deve informar o título do filme e a sala em que será apresentado.
- A partir da listagem apresentada, o funcionário deverá selecionar a sessão desejada pelo cliente.
- Finalmente, o funcionário deverá gerar o ingresso referente à sessão escolhida.

10.32.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

Desenvolva o diagrama de atividade referente ao processo de pagamento de mensalidade para um sistema de clube social, levando em consideração os seguintes fatos:

- Primeiramente deve-se consultar o sócio que deseja pagar mensalidades.
- Após a consulta do sócio, deve-se consultar a(s) mensalidade(s) por ele devida(s).
- Se houver alguma mensalidade em atraso, deve-se calcular os juros referentes ao atraso do pagamento.
- Deve-se então apresentar a(s) mensalidade(s) devida(s) e aguardar que o sócio escolha quais deseja pagar.
- Finalmente, deve-se quitar a(s) mensalidade(s) escolhida(s).

10.32.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Desenvolva o diagrama de atividade referente ao processo de locação de veículo para um sistema de aluguel de veículos, considerando que:

- Ao selecionar a opção de locação de veículos, o sistema deve carregar todos os clientes registrados.

- Em seguida, o sistema deve apresentar todos os veículos disponíveis. A listagem decorrente disto deve mostrar a descrição do automóvel, seu modelo e marca.
- A partir dessa listagem o funcionário deve selecionar o cliente.
- Depois de o cliente ter sido selecionado, deve-se selecionar o automóvel.
- Depois de selecionar o veículo, o funcionário deve inserir os dados da locação, como o período de locação, a que se destina o veículo, para onde o cliente pretende ir etc.
- Finalmente, o funcionário poderá mandar gerar a locação do automóvel selecionado.

10.32.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

Desenvolva o diagrama de atividade referente ao processo de realizar leilão para um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- Ao receber a solicitação do serviço de realizar leilões, o sistema deve carregar todos os leilões ainda não encerrados na interface.
- A partir dessa listagem, o leiloeiro deve selecionar qual leilão deseja iniciar.
- No momento em que um leilão for escolhido para ser iniciado, o sistema precisa carregar todos os itens a serem leiloados nele.
- A partir da listagem dos itens a serem leiloados, o leiloeiro deve escolher um item e anunciá-lo.
- Se houver algum lance para o item anunciado, o sistema deve anunciá-lo e, em seguida, registrá-lo.
- Existe um tempo máximo de espera para que haja lances. Enquanto esse tempo não for atingido, o item permanece sendo anunciado.
- Quando o tempo máximo de espera por um lance for atingido, o processo deve verificar se houve ofertas para o item, caso em que se deve anunciar o participante que ofereceu o maior lance como vencedor. Caso contrário deve-se simplesmente encerrar o anúncio do item.
- Depois de ter sido encerrado o anúncio de um item, deve-se verificar se ainda há itens a anunciar, caso em que o processo passa a anunciar o novo item. Caso contrário, o leilão deve ser encerrado.

10.32.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

Desenvolva o diagrama de atividade referente ao processo de pagamento de diárias para um sistema de controle de hotelaria, de acordo com as seguintes definições:

- No momento em que o hóspede informa o número do quarto para quitar as diárias, o sistema deve consultar o hóspede e todas as diárias devidas pelo aluguel do quarto, apresentando-as ao funcionário.
- A partir dessa listagem, deve-se quitar as diárias apresentadas.
- Se tiver ocorrido a solicitação de quaisquer serviços no período em que o quarto estava ocupado, estes devem ser quitados também;
- O mesmo ocorre se houver quaisquer consumos de frigobar, sendo necessário também quitá-los.

10.33 Solução dos Exercícios

10.33.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

A figura 10.29 apresenta a solução para esse exercício. A seguir explicaremos cada um dos nós de ação desta atividade.

Essa atividade inicia-se com a seleção de todas as sessões de cinema ainda não encerradas. Observe que esta ação apresenta um fluxo de objetos com um objeto da classe Sessao, o que significa que a ação seleciona objetos dessa classe. Após a seleção, a atividade passa à ação que representa o posicionamento sobre o primeiro objeto Sessao selecionado. A seguir é consultada a sala da sessão e em seguida o filme apresentado na mesma. Essas ações poderiam apresentar fluxos de objetos para representar a busca por essas informações, mas preferimos identificar somente as operações mais importantes. Depois de concluídas as consultas, as informações da sessão são apresentadas.

Em seguida a atividade passa a um nó de decisão, onde se deve determinar se ainda há sessões a apresentar, caso em que se passa à ação Posicionar próxima sessão e repetem-se as consultas de sala e filme, apresentando a nova sessão. Se não houver mais sessões, a atividade passa à ação em que aguarda que o funcionário informe a sessão desejada pelo cliente. A seguir, passa-se à ação Gerar ingresso, onde um novo objeto da classe Ingresso será instanciado, como demonstra o fluxo de objetos entre a ação e o objeto. Quando essa última ação for finalizada, a atividade será encerrada.

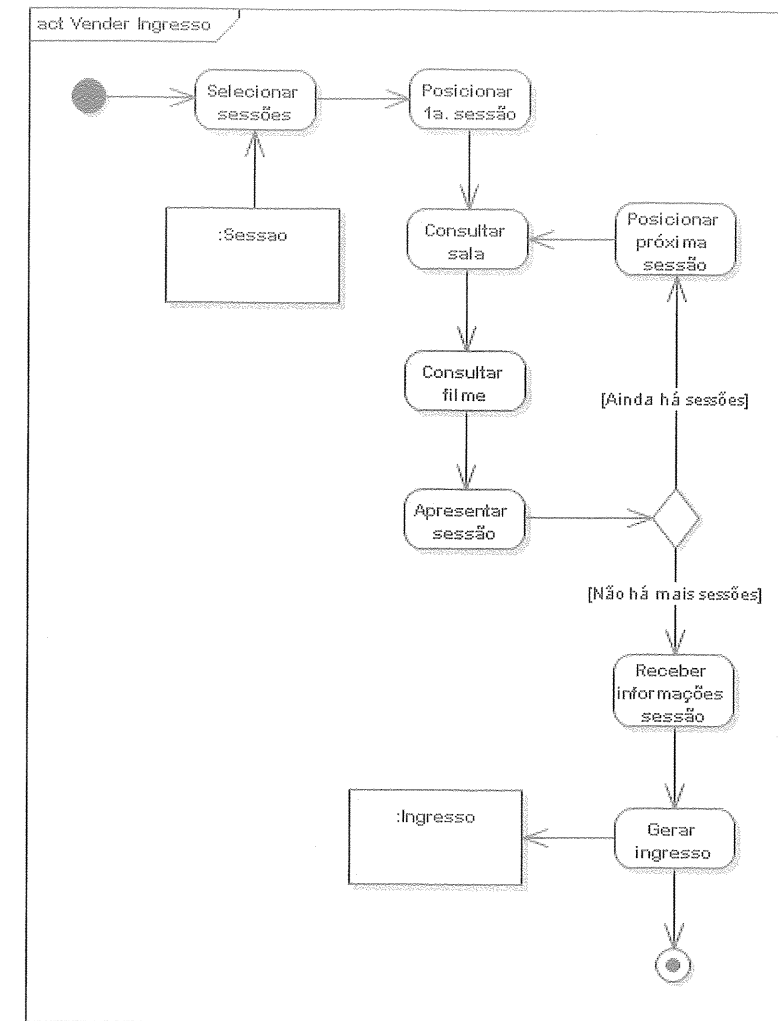


Figura 10.29 – Processo de Venda de Ingressos.

10.33.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

A seguir, por meio da figura 10.30, apresentamos a solução para este exercício.

A primeira ação dessa atividade é caracterizada pelo recebimento do número do cartão do sócio, seguida pela ação em que esse número é consultado. Após isso, a atividade passa para um nó de decisão que testa se o número informado é um número válido e, caso não seja, passa a uma ação que informa que o sócio não foi encontrado, e a atividade é encerrada.

Se o sócio for encontrado, passa-se à ação Selecionar mensalidades do sócio, onde as mensalidades ainda não pagas serão pesquisadas. Observe que existe um fluxo de objetos entre essa ação e um objeto da classe Mensalidade, que indica

que a consulta será feita sobre objetos dessa classe. Como vimos procedendo anteriormente, só identificamos os fluxos de objeto mais importantes, para não deixar o diagrama poluído demais, já que, a rigor, deveria haver um fluxo de objetos durante a ação de consulta de sócio também.

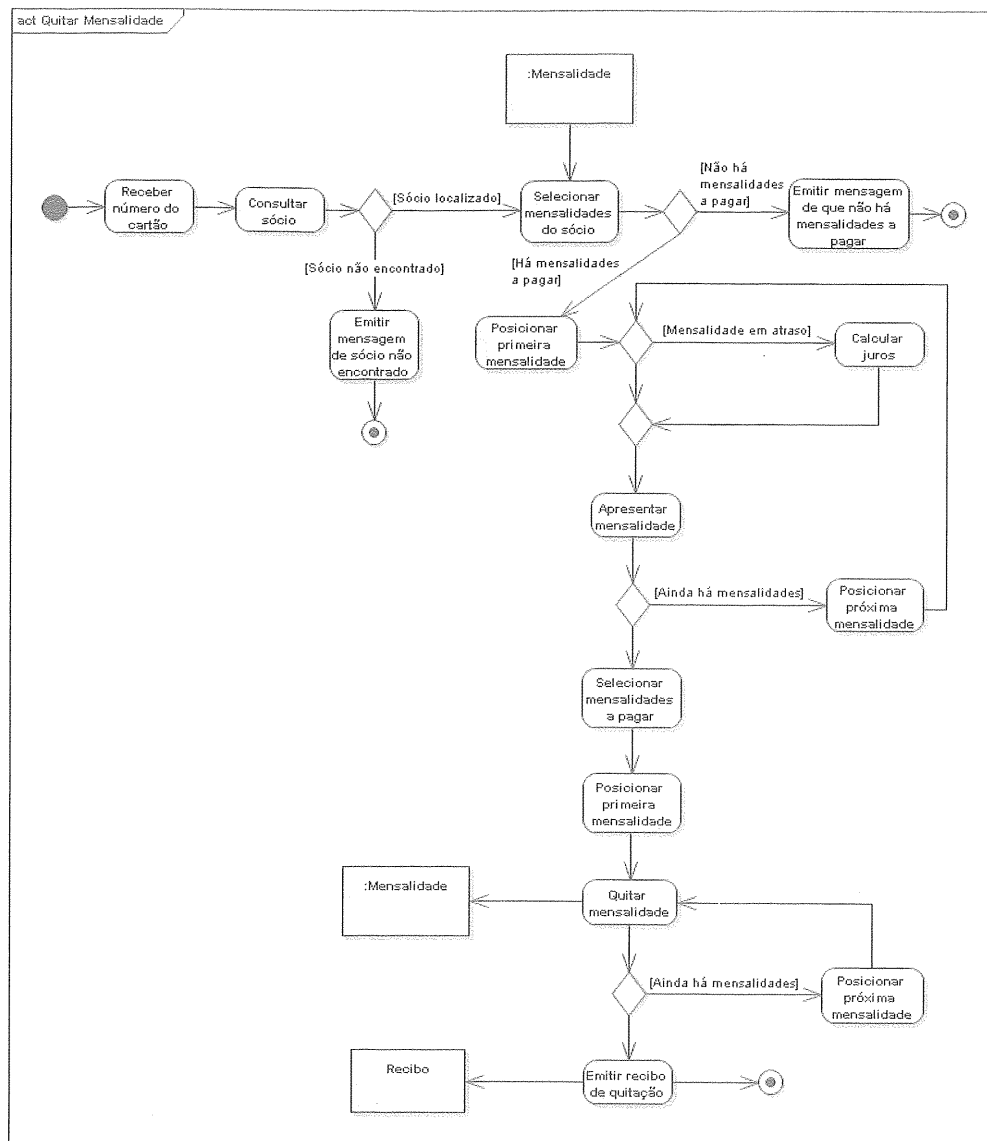


Figura 10.30 – Processo de Pagamento de Mensalidade.

A partir dessa seleção a atividade atinge um novo nó de decisão, que verifica se foi encontrada alguma mensalidade a pagar. Caso não tenha sido encontrada nenhuma mensalidade, deve-se informar isso ao sócio, por meio de um nó de ação, e encerrar a atividade.

Se forem encontradas mensalidades, então a atividade passa ao nó de ação **Posicionar primeira mensalidade** e, em seguida, para um novo nó de decisão, que tem por função verificar se a mensalidade em questão está atrasada, caso em que deverá ser executada a ação **Calcular juros**.

O fluxo dividido pelo nó de decisão é reunido por um nó de união e passa-se à ação que apresenta a mensalidade consultada. Em seguida, é necessário tomar outra decisão, representada por um novo nó de decisão, que verifica se ainda existem mensalidades a apresentar, caso em que executa-se a ação na qual a atividade se posiciona em um novo objeto da classe *Mensalidade* e volta ao nó de decisão que verifica se este está em atraso, repetindo o processo já descrito.

Depois de apresentar as mensalidades, a atividade entra em uma ação em que se aguarda que o sócio selecione as mensalidades que quer pagar. No momento em que o sócio selecioná-las, passa-se à ação **Posicionar primeira mensalidade** e, em seguida, à ação em que a referida mensalidade é quitada. Nessa ação existe um fluxo de objetos para um objeto da classe *Mensalidade*, já que sua situação será alterada e é preciso atualizá-lo.

Em seguida a atividade encontra um novo nó de decisão, que verifica se existem mais mensalidades a quitar, caso em que é executada a ação **Posicionar próxima mensalidade** e volta-se à ação de quitar mensalidade. Caso contrário, a atividade executa a ação **Emitir recibo de quitação** e, após o término desta, encerra-se a atividade. Observe que há novamente um fluxo de objetos, atingindo um objeto chamado *Recibo*. Esse objeto não existe no modelo de classes e representa apenas o recibo físico que o sistema deve emitir para o cliente.

10.33.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Apresentaremos aqui a atividade referente à solução desse exercício, ilustrado pela figura 10.31.

O primeiro passo desta atividade é representado pela ação **Selecionar clientes**, que, como o leitor pode perceber, apresenta um fluxo de objetos com um objeto da classe *Cliente*, determinando que a busca por clientes será feita sobre objetos dessa classe. Em seguida, a atividade passa à ação **Posicionar primeiro cliente**, para depois apresentá-lo na interface.

A seguir, a atividade encontra um nó de decisão, o qual verifica se ainda há clientes a apresentar e, em caso positivo, executa a ação **Posicionar próximo cliente** e volta à ação que apresenta o cliente, repetindo esse laço enquanto houver clientes a apresentar.

Quando não houver mais clientes, a atividade seleciona todos os automóveis disponíveis. Essa consulta é feita em objetos da classe *Automóvel*, como demonstra o fluxo de objetos. A atividade passa então para a ação *Posicionar primeiro automóvel*, ou seja, selecionará um dos objetos da classe *Automóvel* pesquisados. Depois disso, a atividade executará as ações para consultar o modelo e a marca do automóvel em questão, o que poderia gerar fluxo de objetos, mas não achamos necessário detalhar tanto.

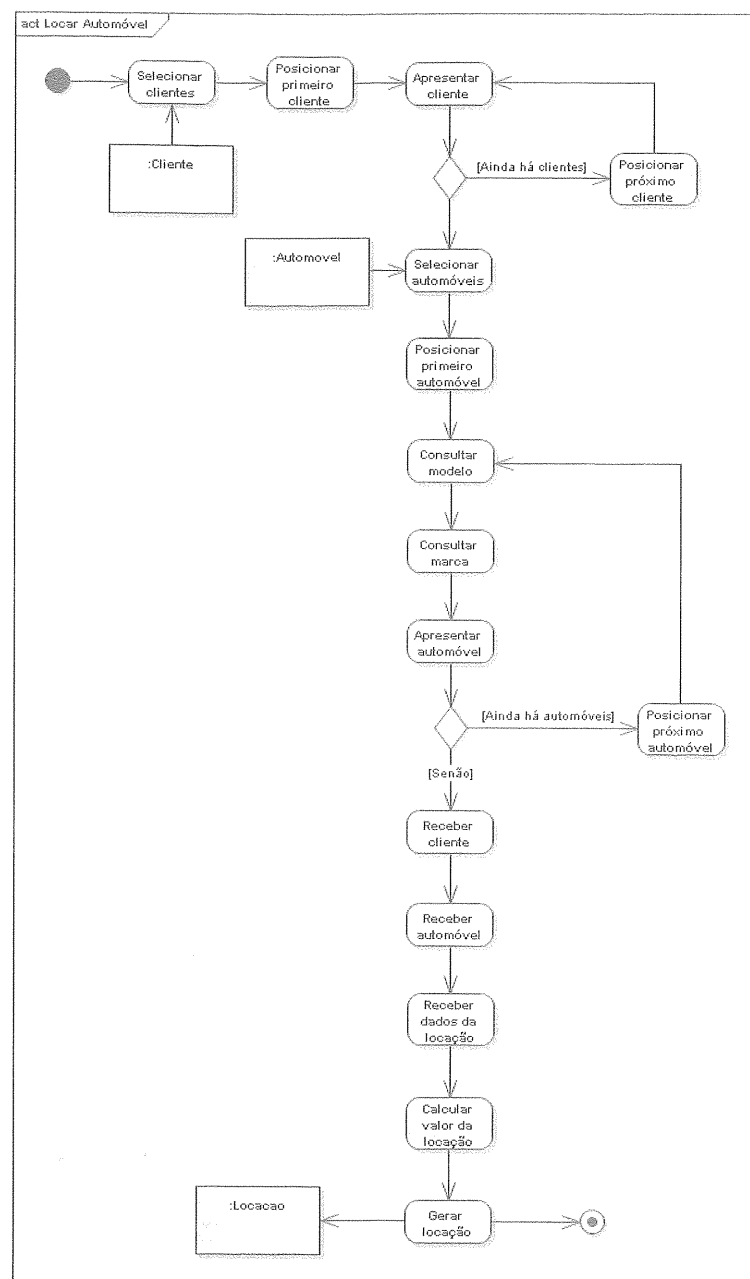


Figura 10.31 – Processo de Locação de Veículo.

Em seguida a atividade passa a um novo nó de decisão, que testa se ainda há automóveis e, em caso positivo, posiciona sobre o próximo automóvel, repetindo a consulta de modelo e marca e o apresentando. Esse laço será repetido enquanto houver automóveis.

A partir dessa listagem o funcionário deve selecionar o cliente, o que é representado pela ação *Receber cliente*. Em seguida deve-se selecionar o automóvel, o que está representado pela ação *Receber automóvel* e, após isso, o funcionário deve inserir os dados da locação, como demonstra a ação *Receber dados da locação*.

Finalmente, a atividade passa à ação final, onde é gerada a locação do veículo selecionado. Essa ação causa um fluxo de objetos para um objeto da classe *Locacao*, representando sua instanciação. Após a conclusão dessa ação, a atividade é encerrada.

10.33.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

A atividade que soluciona esse exercício está contida na figura 10.32. Essa atividade inicia-se com a ação *Selecionar leilões*, que, como demonstra o fluxo de objetos, seleciona a partir de objetos da classe *Leilao* todos os leilões ainda não encerrados. Em seguida passa-se à atividade *Posicionar primeiro leilão*, onde é selecionado o primeiro objeto *Leilao* pesquisado. O leilão em questão é apresentado na ação seguinte e depois há um teste, representado por um nó de decisão, que verifica se ainda há leilões. Em caso positivo, a atividade posiciona-se sobre o próximo leilão, apresenta-o, e testa novamente se ainda há leilões, ficando nesse laço enquanto a condição for verdadeira.

Quando não houver mais leilões, a atividade aguarda que o leiloeiro escolha um dos leilões apresentados. No momento em que um leilão for escolhido é executada a ação *Iniciar leilão*, que como o nome diz, inicia um novo leilão. Essa ação leva à ação seguinte, onde são selecionados os itens a serem leiloados no leilão escolhido. O fluxo de objetos associado a essa última ação demonstra que é feita uma pesquisa nos objetos da classe *Item_Leilao* associados ao leilão selecionado.

Em seguida, a atividade envolve-se em um laço, onde serão apresentados os itens referentes ao leilão. Primeiramente é executada a ação *Posicionar primeiro item*, onde o primeiro dos itens pesquisados é selecionado. Em seguida a atividade apresenta o item em questão e testa, por meio de um nó de decisão, se ainda existem itens a apresentar, caso em que ela se posiciona no próximo item e repete a operação.

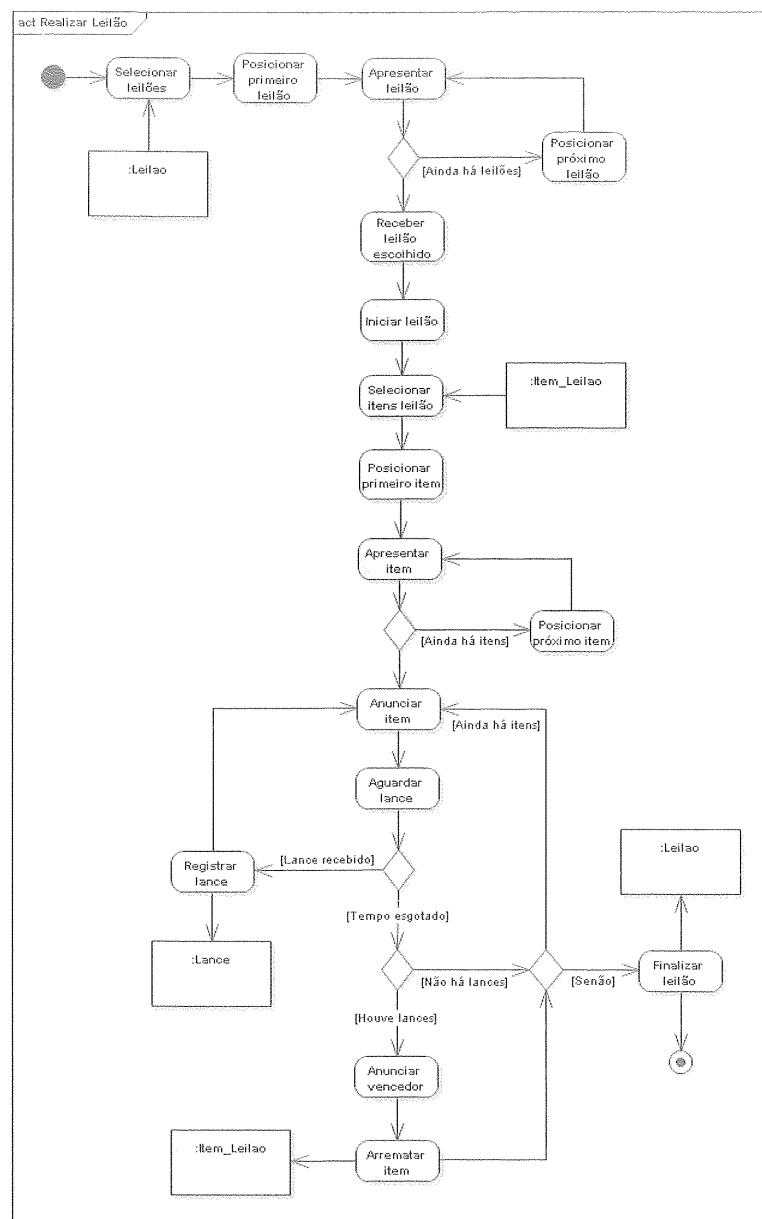


Figura 10.32 – Processo de Realizar Leilão.

A partir da listagem dos itens a serem leiloados, o leiloeiro deve escolher um item a leiloar e anunciá-lo, o que é representado pela ação **Anunciar item**. Em seguida, deve-se aguardar por um período as possíveis ofertas para o item anunciado. Isso é representado pela ação **Aguardar lance**. A partir daí a atividade atinge um nó de decisão, no qual se deve verificar a ocorrência de uma entre duas possibilidades.

A primeira possibilidade representa o recebimento de um lance por parte de um dos participantes do leilão. Nessa situação, a atividade executa a ação **Registrar lance**, onde é instanciado um novo objeto da classe **Lance**, como demonstra o fluxo de objetos. Em seguida a atividade volta a anunciar o item e fica aguardando novas ofertas.

A segunda possibilidade representa o fim do tempo máximo de espera para que haja lances. Quando essa situação ocorre passa-se a um segundo nó de decisão, que deve testar se o item recebeu ofertas ou não. Caso tenha recebido lances, a atividade passa à ação que anuncia o vencedor e, em seguida, à ação que define o item como arrematado. Essa ação altera a situação do objeto **Item_leilao** selecionado. Por esse motivo há um fluxo de objetos entre a ação e esse objeto. A seguir, a atividade passa a um novo nó de decisão, que verifica se ainda há itens a leiloar. Esse nó de decisão também é atingido caso não tenham ocorrido lances para o item.

Se ainda houver itens a leiloar a atividade anuncia o item seguinte e repete todo o processo já explicado. Caso não haja mais itens a leiloar, a atividade executa a última ação, que é responsável por finalizar o leilão. Essa ação marca o leilão selecionado como encerrado, por esse motivo há um fluxo de objetos entre ela e um objeto da classe **Leilao**. Após a execução dessa ação, a atividade é encerrada.

10.33.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

Aqui apresentamos a solução para esse exercício por meio de um diagrama de atividade representado na figura 10.32.

Ao examinarmos essa figura percebemos que ela apresenta três atividades, sendo a principal justamente a de quitar diárias. As outras duas são atividades complementares desse processo. Assim, iniciaremos a explicar esse diagrama a partir da atividade **Quitar Diárias**.

Essa atividade inicia-se no momento em que o funcionário fornece o número do quarto a ser consultado. Depois dessa ação, a atividade passa à ação de consultar o quarto informado, passando depois à ação de consultar o hóspede que o aluga e, em seguida, à ação de selecionar as diárias devidas. Observe que essa consulta é realizada sobre objetos da classe **Diaria**, como demonstra o fluxo de objetos entre a ação e o objeto da classe **Diaria**.

Após isso, a atividade executa a ação **Posicionar primeira diária**, onde é selecionada a primeira diária pesquisada. A ação seguinte apresenta a referida diária e, a seguir, passa-se a um nó de decisão que deve verificar se ainda há diárias a apresentar, caso em que a atividade executa a ação **Posicionar próxima diária** e volta à ação que apresenta a diária, ficando nesse laço enquanto houver diárias a apresentar.

Quando não houver mais diárias a apresentar, a atividade passa à ação de **Receber diárias a quitar**, onde o funcionário informa quais diárias deverão ser quitadas. Assim, a atividade executa a ação de **Posicionar primeira diária** e, em seguida, entra em um segundo laço no qual será executada a ação que quitará a diária selecionada, definido esse objeto como quitado, como mostra o fluxo de objetos. É feito um teste por meio de um nó de decisão que verifica se ainda há diárias a quitar. Em caso positivo, a atividade se posiciona na próxima diária e repete a operação de quitação; caso contrário, a atividade é encerrada.

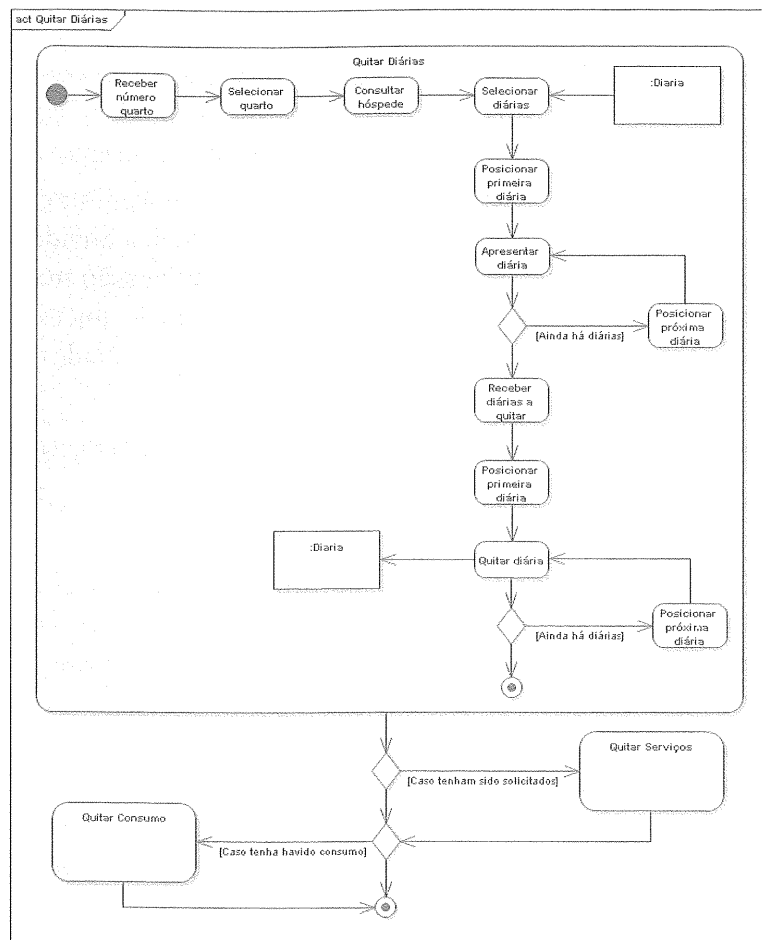


Figura 10.33 – Processo de Pagamento de Diárias.

Após a conclusão dessa atividade, é realizado um teste para determinar a necessidade de execução da atividade **Quitar Serviços**, caso algum dos serviços oferecidos pelo hotel tenham sido solicitados. Após esse teste é realizado outro para determinar se o hóspede realizou algum consumo de frigobar, caso em que deverá ser executada a atividade **Quitar Consumo**. Após isso o processo é encerrado.

CAPÍTULO 11

Diagrama de Visão Geral de Interação

Este diagrama é uma variação do diagrama de atividade. Seu objetivo é fornecer uma visão geral do controle de fluxo. O diagrama utiliza quadros no lugar dos nós de ação, embora símbolos como o nó de decisão e o nó inicial sejam também utilizados. Existem basicamente dois tipos de quadros: quadros de interação, que contêm qualquer tipo de diagrama de interação da UML, e quadros de ocorrência de interação, que normalmente fazem uma referência a um diagrama de interação, mas não apresentam seu detalhamento. A figura 11.1 apresenta um pequeno exemplo de diagrama de visão geral de interação representando o processo geral de encerramento de conta especial referente ao sistema de controle bancário que temos modelado neste livro.

O leitor notará que esse diagrama é muito semelhante ao diagrama de sequência referente ao mesmo processo, já explicado no capítulo 7, porém, apresentando uma visão um pouco diferente. Isso ocorre porque o sistema de controle bancário não apresenta muitos processos interligados em um processo geral, sendo o processo de encerramento de conta o que mais se adéqua aos objetivos desse diagrama.

Nesse exemplo estão representados três quadros de ocorrência de interação, referentes aos processos de **Emissão de Saldo**, **Realizar Saque** e **Realizar Depósito**, e um quadro de interação referente ao processo de **Encerrar Conta**, que contém um diagrama de sequência. Na verdade os quadros de ocorrência de interação também se referem a diagramas de sequência. No entanto, optamos por apenas referenciá-los para não tornar o diagrama grande demais, o que é muito comum nesse tipo de diagrama. Também é recomendável, quando se vai inserir quadros de interação, procurar representar somente o fluxo mais geral ou importante, bem como os objetos mais essenciais, para não deixar o diagrama grande demais.

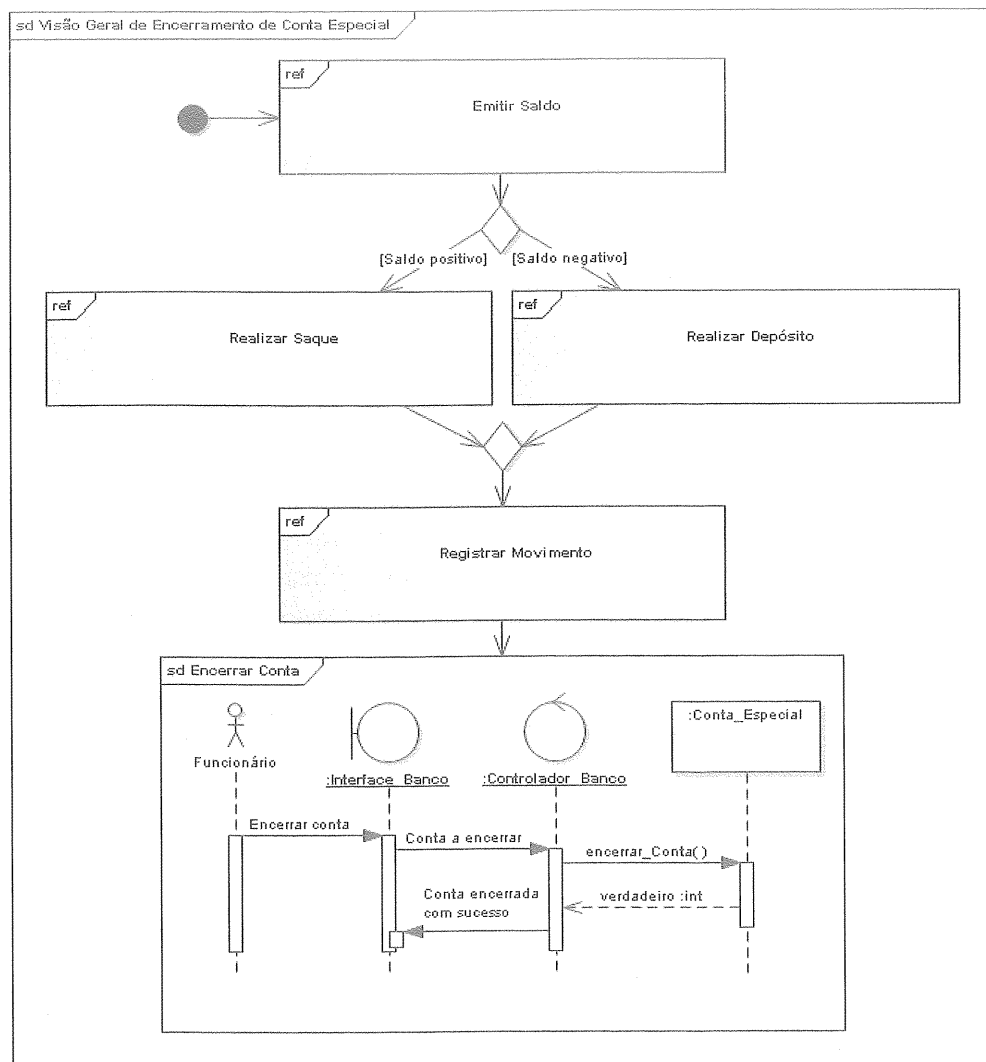


Figura 11.1 – Diagrama de Visão Geral de Interação – Processo Geral de Encerramento de Conta Especial.

Nesse diagrama o fluxo se inicia com a execução do processo de emissão de saldo. Depois de esse ter sido concluído, passa-se a um nó de decisão que verifica se o saldo da conta é positivo ou negativo. Caso seja positivo, é executado o processo de **Realizar Saque**, e se for negativo, o processo de **Realizar Depósito**. Um nó de união une os dois fluxos divididos anteriormente. Em seguida é necessário registrar o movimento, tenha ele sido referente a um saque ou a um depósito, conforme demonstra o quadro de ocorrência de interação **Registrar Movimento**.

O processo se encerra após a execução do quadro de interação **Encerrar Conta**, onde o funcionário solicita o encerramento da conta à interface, que repassa a

solicitação à controladora e que, por sua vez, dispara o método **Encerrar_Conta** em um objeto da classe **Conta_Especial**. A execução desse método retornará verdadeiro se o método for concluído com sucesso, o que fará com que a controladora mande a interface exibir a mensagem de “Conta encerrada com sucesso”.

11.1 Exemplo de Diagrama de Visão Geral de Interação – Processo Geral de Conclusão de Pedido – Sistema de Livraria Digital

Esse exemplo (Figura 11.2) baseia-se em um sistema de livraria virtual, cujo diagrama de casos de uso pode ser visto na figura 3.7 do capítulo sobre o diagrama de casos de uso. Nesse exemplo um cliente pode adicionar livros, logar, visualizar o carrinho e concluir o pedido, sendo esse o processo final para a conclusão da compra. Todas essas funcionalidades são representadas aqui como quadros de ocorrência de interação. Aqui demonstramos o processo geral que um cliente deve percorrer para comprar livros. Esse exemplo está um pouco simplificado, mas o objetivo desse diagrama é dar uma visão geral do processo.

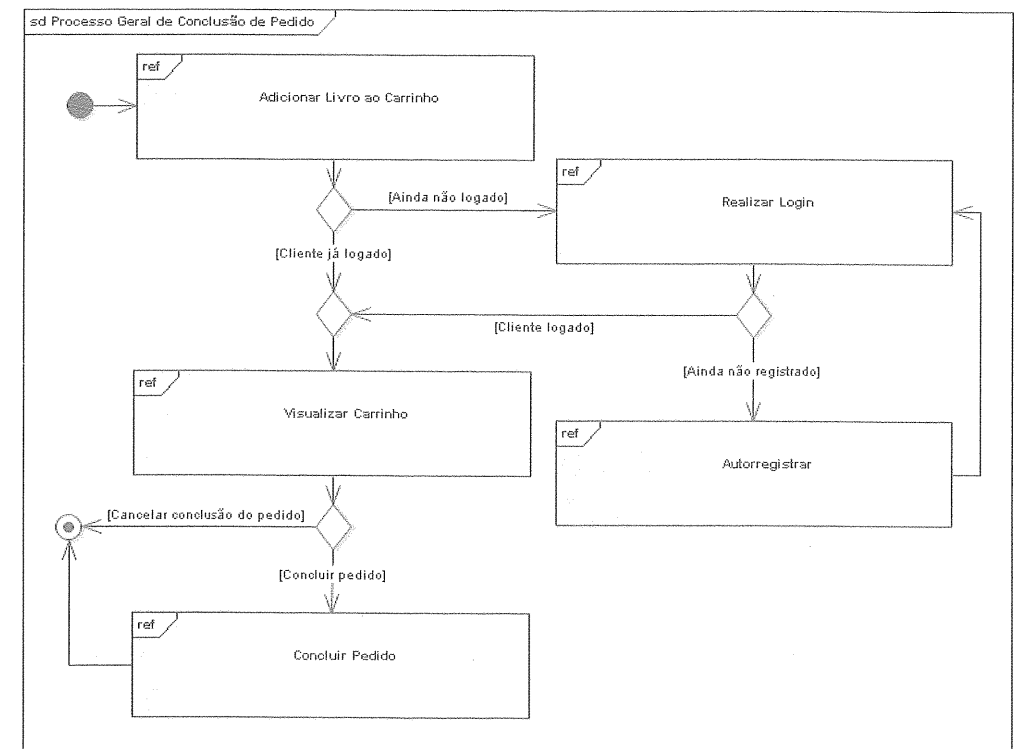


Figura 11.2 – Processo Geral de Conclusão de Pedido – Sistema de Livraria Digital.

Nesse exemplo, o cliente inicia pelo processo de Adicionar Livro ao Carrinho, onde ele escolherá e adicionará quantos livros quiser. Quando o cliente desejar concluir o pedido, o sistema deve antes realizar um teste, representado aqui por um nó de decisão, onde se deve verificar se o cliente já está logado.

Se o cliente já estiver logado, passa-se ao processo de Visualização do Carrinho, porém, se o cliente não tiver se logado ainda, é necessário executar o processo de Realizar Login, onde o cliente se logará no sistema. Pode-se notar que há um segundo teste representado por um segundo nó de decisão, que verifica se o cliente está registrado no sistema. Se estiver registrado, passa-se ao processo de Visualização de Carrinho (observe que o fluxo dividido anteriormente é unido por um nó de união), mas se o cliente não estiver registrado, deve-se executar o processo de Autorregistrar, onde o cliente poderá se cadastrar no sistema.

Durante o processo de Visualização de Carrinho, será apresentada ao cliente a lista de todos os livros que ele adicionou, bem como suas quantidades, e será permitido a este aumentar ou diminuir a quantidade de qualquer livro, bem como excluir qualquer um deles.

Após a conclusão desse processo deve-se realizar um último teste, uma vez que o sistema deve perguntar se o cliente quer realmente concluir o pedido. O cliente pode nesse momento cancelar a conclusão do pedido ou, se realmente desejar, executar o processo Concluir Pedido.

11.2 Exercícios Propostos

Aqui daremos continuidade à modelagem dos sistemas que temos modelado como exercício ao longo do livro, porém, nem todos os sistemas farão parte desta seção, uma vez que nem todos se enquadram no escopo desse diagrama.

11.2.1 Sistema de Controle de Clube Social – Processo Geral de Associação

Desenvolva o diagrama de visão geral de interação referente ao processo geral de associação para um sistema de clube social, sabendo que:

- Primeiramente o candidato a sócio deve percorrer os passos do processo para apresentar um pedido de aceitação.
- Caso o pedido seja recusado, o candidato não será aceito, porém, se seu pedido for aprovado, então este passará ao processo por meio do qual irá se associar no clube.
- Se o sócio possuir dependentes, deverá passar ao processo de associação de dependentes.

11.2.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estadia

Desenvolva o diagrama de visão geral de interação referente ao processo geral de encerramento de estadia para um sistema de controle de hotel, levando em consideração os seguintes requisitos:

- O processo de encerramento de estadia exige, antes de mais nada, que todas as diárias ainda devidas pelo hóspede sejam pagas.
- É necessário também verificar se houve solicitação de algum dos serviços oferecidos pelo hotel, que deverão ser pagos também.
- Igualmente é preciso verificar se houve consumo do frigobar, que deverá ser quitado da mesma forma.
- Somente depois de essas etapas serem concluídas será possível executar o processo de encerramento de conta propriamente dito.

11.3 Solução dos Exercícios

11.3.1 Sistema de Controle de Clube Social – Processo Geral de Associação

A solução para esse exercício é apresentada na figura 11.3. Nesse exercício são representados dois quadros de ocorrência de interação, referentes aos processos de Apresentar Pedido de Aceitação e Associar Dependentes, e um quadro de interação referente ao processo de Associar Sócio, que contém um diagrama de sequência.

O diagrama se inicia com a execução do processo Apresentar Pedido de Aceitação, onde um candidato a sócio irá apresentar formalmente um pedido para ser aceito no clube, e este verificará se o candidato é adequado para ser sócio. A seguir, passa-se a um teste, representado por um nó de decisão, onde, caso o pedido tenha sido recusado, encerra-se o processo.

Caso o pedido seja aceito, passa-se ao processo de Associar Sócio, representado por um quadro de interação, onde o sócio informa seus dados pessoais através da interface e confirma. Essas informações são repassadas à controladora, que dispara o método regSocio em um objeto da classe Socio, instanciando-o. Esse método retorna um long (caso seja concluído com sucesso) representando o número do cartão do novo sócio. De posse desse número, a controladora solicita à interface que emita o cartão do novo sócio.

Após o término desse processo, é feito um novo teste, onde é necessário verificar se o sócio possui dependentes. Caso não possua, o processo geral encerra-se nesse momento. Porém, caso haja dependentes, é necessário ainda executar o processo Associar Dependentes antes da conclusão do processo geral.

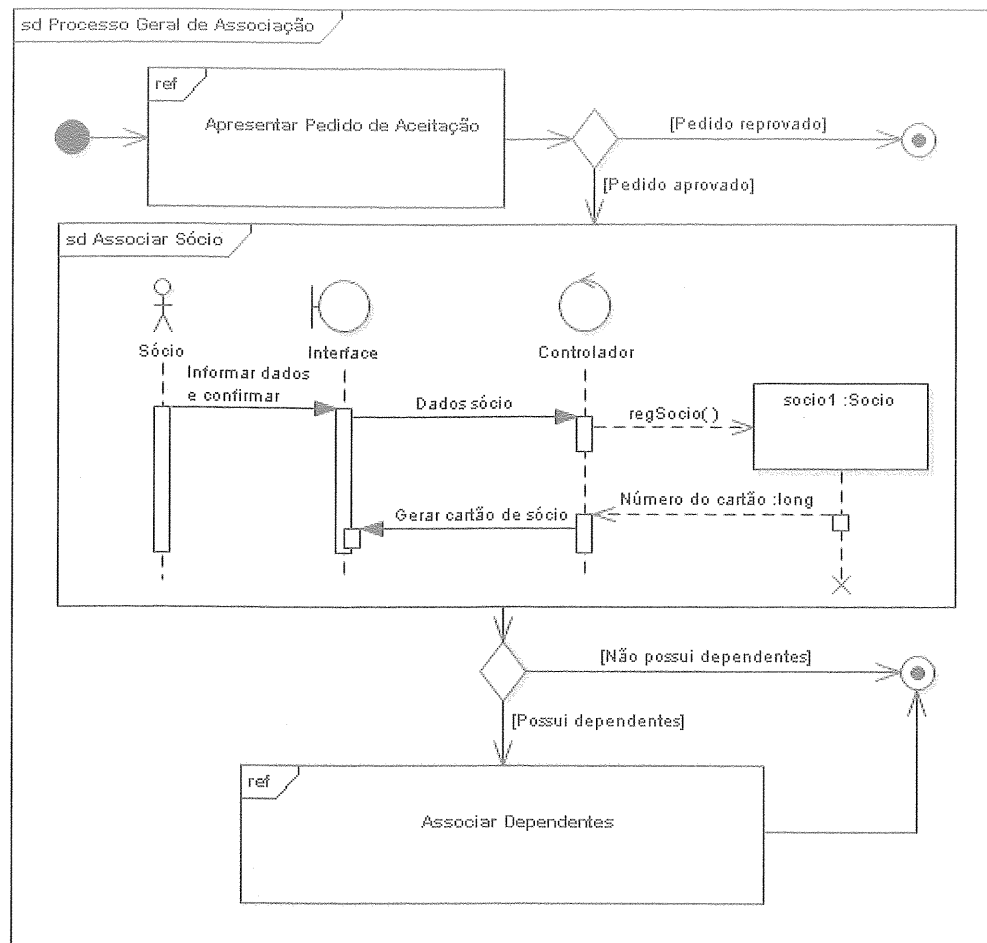


Figura 11.3 – Diagrama de Visão Geral de Interação – Processo Geral de Associação.

11.3.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estadia

A figura 11.4 apresenta a solução para esse exercício. É recomendável examinar o diagrama de casos de uso da figura 3.29, referente ao sistema de controle de hotel, onde o leitor notará a existência de uma associação do tipo <<include>> entre os casos de uso Encerrar Estadia e Quitar Diárias. Existem ainda duas associações de extensão entre o caso de uso Quitar Diárias e os casos de uso Quitar Serviços e Quitar Consumo. Sendo assim, antes de encerrar a estadia é preciso quitar todas as diárias ainda devidas e, se tiver ocorrido a solicitação de serviços ou consumo do frigobar, estes deverão também ser pagos para somente então poder encerrar a estadia. Essas associações serão representadas, sob outra visão, nessa solução.

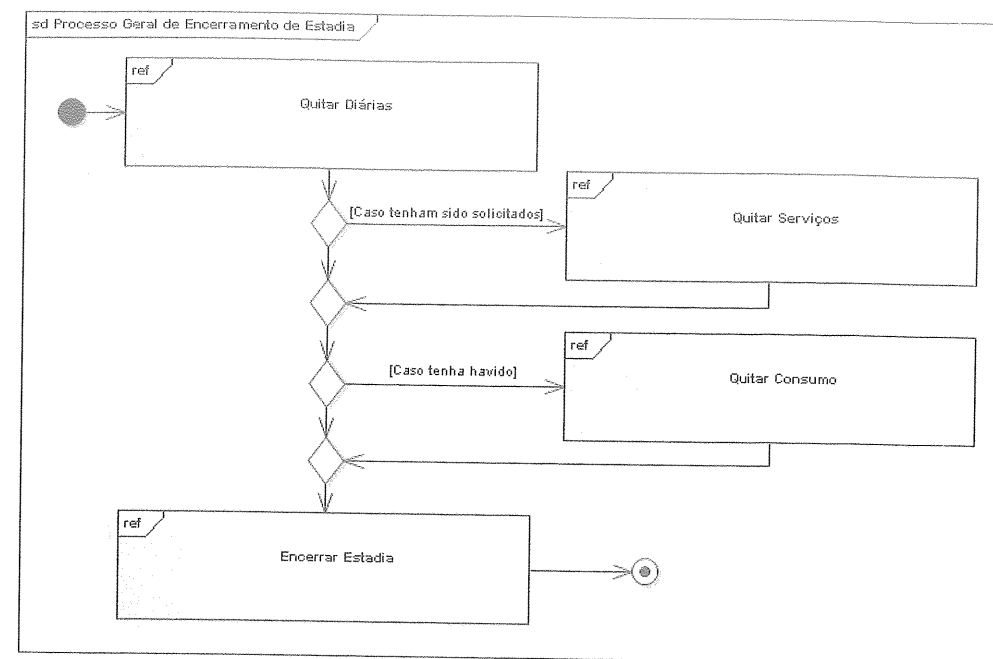


Figura 11.4 – Processo Geral de Encerramento de Estadia – Sistema de Controle de Hotel.

Pelos motivos já explicados antes, o primeiro processo a ser iniciado nesse diagrama é o processo de Quitar Diárias. Quando este for finalizado é necessário fazer um teste para verificar se houve solicitação de algum serviço. Em caso positivo, deve-se executar o processo de Quitar Serviços antes de passar ao teste seguinte.

Observe que o fluxo dividido pelo nó de decisão anterior é reunido por um nó de união, o qual conduz a outro nó de decisão, cuja função é verificar se houve consumo do frigobar. Se isso for verdadeiro, deverá ser executado o processo de Quitar Consumo.

Depois de terem sido quitadas as diárias e possivelmente os serviços e consumos, finalmente passa-se ao processo de encerramento propriamente dito, onde a situação do quarto será definida como liberado, e o hóspede estará quite com o hotel. Após a conclusão desse último processo, o processo como um todo será encerrado.

CAPÍTULO 12

Diagrama de Componentes

O diagrama de componentes, como seu próprio nome diz, identifica os componentes que fazem parte de um sistema, um subsistema ou mesmo os componentes ou classes internas de um componente individual. Um componente pode representar tanto um componente lógico (um componente de negócio ou de processo) ou um componente físico como arquivos contendo código-fonte, arquivos de ajuda (help), bibliotecas, arquivos executáveis etc.

O diagrama de componentes pode ser utilizado como uma forma de documentar como estão estruturados os arquivos físicos de um sistema, permitindo assim uma melhor compreensão do mesmo, além de facilitar a reutilização de código. Esse diagrama também pode identificar os componentes utilizados no desenvolvimento de sistemas baseado em componentes.

12.1 Componente

Um componente pode sempre ser considerado uma unidade autônoma dentro de um sistema ou subsistema. Ele pode conter uma ou mais interfaces fornecidas e requeridas (potencialmente expostas via portas), e seus interiores são transparentes e inacessíveis por outro meio que não seja fornecido por suas interfaces. Embora possa ser dependente de outros elementos em termos de interfaces requeridas, um componente é encapsulado, e suas dependências são projetadas de tal forma que ele possa ser tratado tão independentemente quanto possível.

Na UML 1.x um componente era representado por um retângulo com dois retângulos menores sobressaindo-se à sua esquerda. A partir da UML 2 esse símbolo foi substituído por um retângulo contendo internamente o antigo símbolo, conforme demonstra a figura 12.1.



Figura 12.1 – Exemplo de Componente.

O componente **Gerenciador de Contas** apresentado na figura 12.1 representa um módulo executável, responsável pelo gerenciamento e pela manutenção de contas bancárias. Isso pode ser mais bem-explicitado pelo uso de estereótipos, os quais, como já explicado anteriormente, atribuem características extras a um determinado item de um diagrama. A UML prevê diversos estereótipos prontos para o diagrama de componentes, tais como:

- **Executável** (executable) – determina que o componente em questão é um arquivo executável, ou seja, um arquivo já compilado que pode executar uma série de instruções. O termo executável é genérico, podendo se referir a módulos compilados em Java, que às vezes não são executáveis no sentido clássico, sendo interpretados pela máquina virtual.
- **Biblioteca** (library) – refere-se a bibliotecas contendo funções e sub-rotinas que podem ser compartilhadas por diversos componentes executáveis, podendo estas ser fornecidas pela própria linguagem em que o sistema será desenvolvido ou ser criadas pelos desenvolvedores do sistema.
- **Tabela** (table) – refere-se a repositórios físicos de dados, onde as informações produzidas pelo sistema deverão ser armazenadas.
- **Documento** (document) – faz referência a arquivos-texto utilizados por outros componentes do sistema, como arquivos de ajuda (help), por exemplo.
- **Arquivo** (file) – pode se referir a qualquer outro arquivo que componha o sistema, como arquivos de código-fonte dos módulos do sistema, por exemplo.

A figura 12.2 apresenta exemplos do uso de estereótipos em componentes.

Podemos perceber, pelo exame dos estereótipos representados na figura, que o componente **Gerenciador de Contas** refere-se a um módulo executável, que o componente **Contas** refere-se a uma tabela de um banco de dados e que o componente **Ajuda** é um documento (no caso um arquivo de help). Observe que esse último estereótipo é gráfico, contendo o desenho de um documento em seu lado direito.

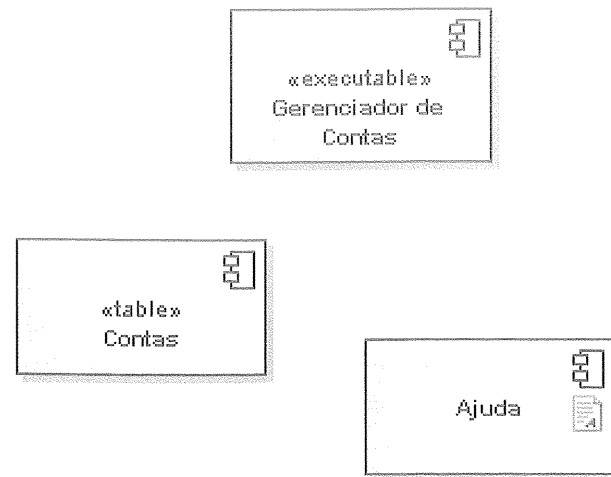


Figura 12.2 – Componentes com Estereótipos.

12.2 Interfaces Fornecidas e Requeridas

Na UML 1.x uma das principais formas de representar comunicação entre os componentes era realizada por meio do relacionamento de dependência, já explicado no capítulo sobre o diagrama de classes. No entanto, a partir da UML 2, embora esse tipo de relacionamento continue sendo válido e utilizado, passou-se a utilizar com muito mais frequência interfaces fornecidas e requeridas, as mesmas descritas no diagrama de classes, conforme demonstra a figura 12.3.

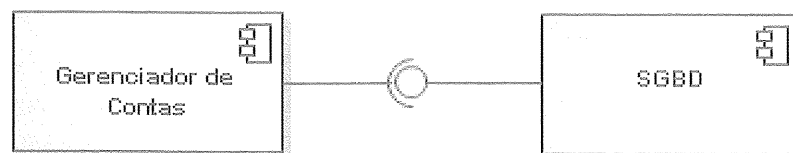


Figura 12.3 – Relacionamento entre Componentes por meio de Interfaces Fornecidas e Requeridas.

Nesse exemplo podemos perceber que o componente Gerenciador de Contas tem um relacionamento com o componente SGBD, que representa um sistema gerenciador de banco de dados. Observe que o componente Gerenciador de Contas tem uma interface requerida com o componente SGBD, e este, uma interface fornecida com o componente Gerenciador de Contas, uma vez que este último componente solicita frequentemente a gravação e recuperação de informações no SGBD.

12.3 Classes e Componentes Internos

Um componente pode conter ou implementar uma ou mais classes internas, podendo conter também componentes internos. A representação de um componente sem apresentar seus componentes ou classes internas é chamada de visão de caixa preta. Já a representação de um componente com suas classes ou componentes internos é chamada visão de caixa branca. A contenção de classes por um componente significa que este as implementa de alguma maneira.

A figura 12.4 demonstra um exemplo de componente com classes internas. O leitor notará que o componente Gerenciador de Contas implementa as classes necessárias para a manutenção de contas do sistema bancário. É possível também definir as associações entre os componentes internos, mas isso pode deixar o componente exageradamente grande.

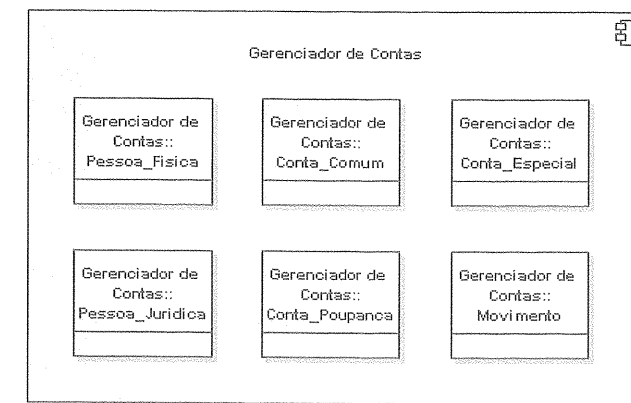


Figura 12.4 – Componente com Classes Internas.

Outra forma de representar as classes internas de um componente é por meio do relacionamento de dependência, conforme demonstra a figura 12.5.

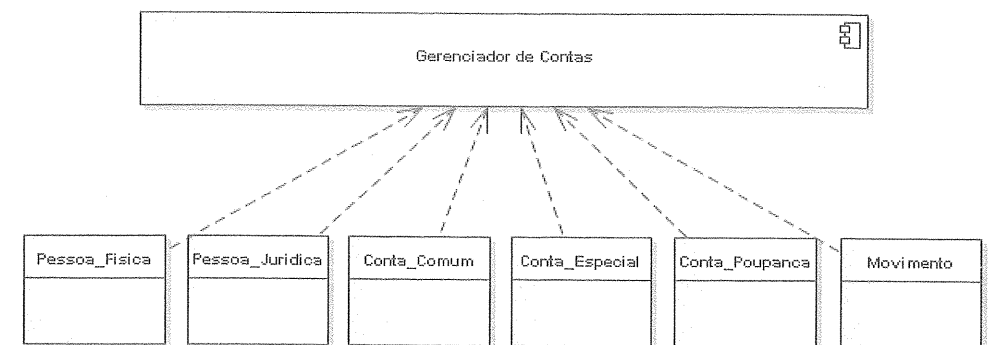


Figura 12.5 – Classes Internas relacionadas a um componente por meio de Dependências.

12.3.1 Portas

É comum o uso de portas para comunicar os elementos internos de um componente com o ambiente externo quando estes solicitam ou executam algum serviço, conforme demonstra a figura 12.6.

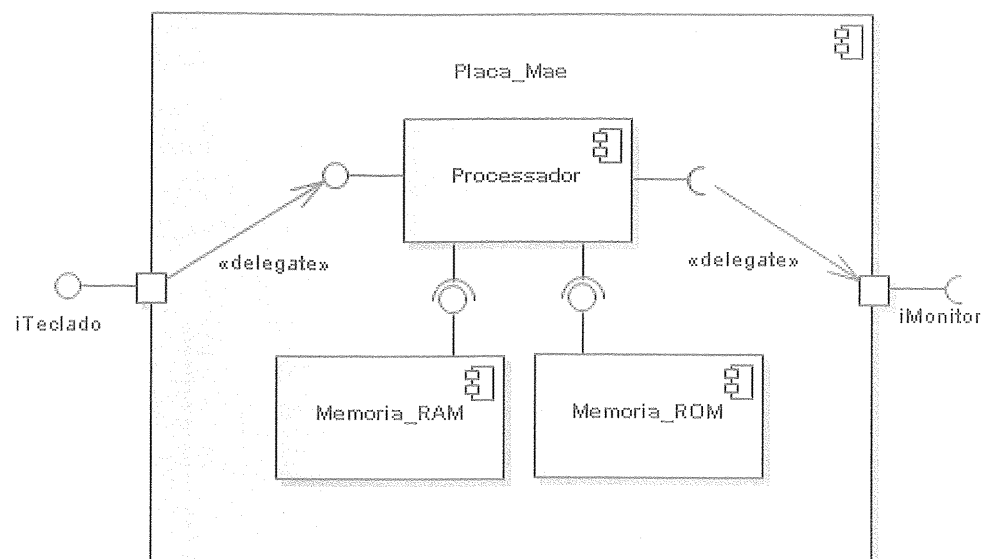


Figura 12.6 – Visão de Caixa Branca de um Componente.

Nesse exemplo representamos um componente chamado *Placa_Mae* com seus componentes internos. Observe que o componente *Processador* solicita serviços dos componentes *Memoria_RAM* e *Memoria_ROM*, além de se comunicar com o ambiente externo por meio de portas. Observe o relacionamento de dependência com o estereótipo <<delegate>>, significando que o serviço atribuído ao componente *Placa_Mae* está sendo delegado a seu componente interno *Processador*. Essa mesma abordagem pode ser também utilizada com as classes internas de um componente.

12.4 Exemplo de Diagrama de Componentes – Sistema de Controle Bancário

A seguir apresentaremos o diagrama de componentes equivalentes aos módulos executáveis do sistema de controle bancário que estamos modelando ao longo deste livro. Alguns módulos não são exatamente executáveis, como o componente que representa a interface do caixa eletrônico, e outros podem não pertencer exclusivamente ao sistema como os componentes que representam o firewall e o sistema gerenciador de banco de dados, mas são indispensáveis para o funcionamento do mesmo.

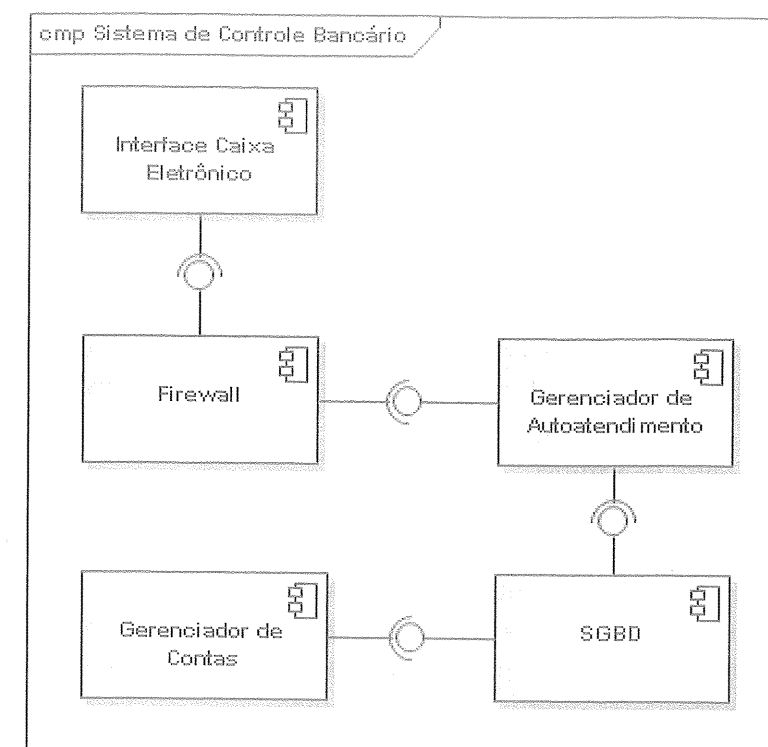


Figura 12.7 – Diagrama de Componentes – Sistema de Controle Bancário.

Esse diagrama representa os seguintes componentes:

■ Interface Caixa Eletrônico

Esse componente representa simplesmente a interface do caixa eletrônico, por meio da qual o cliente pode solicitar serviços como emissão de saldo, emissão de extrato, saque ou depósito. Observe que esse componente apresenta uma interface requerida com o componente firewall, uma vez que ele solicita ao firewall a transmissão dos eventos ocorridos sobre a interface ao Gerenciador de Autoatendimento.

■ Firewall

Esse componente representa um software que implementa um sistema de firewall, ou seja, é um software de segurança que tenta impedir que usuários não autorizados invadam o sistema. Esse componente não faz realmente parte do sistema bancário, no entanto, por medida de segurança, todo o tráfego externo deve passar por ele antes de se comunicar com o sistema propriamente dito.

■ Gerenciador de Autoatendimento

Esse componente representa o módulo do sistema responsável por gerenciar as solicitações realizadas pelos clientes na interface do caixa eletrônico. Ele deve interpretar os eventos ocorridos na interface e solicitar as execuções adequadas a eles.

■ SGBD

Esse componente representa o sistema gerenciador de banco de dados com o qual o sistema interage. É responsável por persistir e recuperar os dados do sistema. Não faz realmente parte do sistema, mas é indispensável a ele.

■ Gerenciador de Contas

Esse é o módulo responsável por gerir as contas mantidas pela instituição bancária. É por meio dele que é possível abrir ou encerrar contas ou emitir relatórios gerenciais. É o núcleo do sistema propriamente dito. Observe que tanto esse módulo como o **Gerenciador de Autoatendimento** contêm interfaces requeridas com o SGBD, uma vez que este executa os pedidos de gravação e recuperação das informações do sistema.

12.5 Exercícios Propostos

Como já tem ocorrido nos capítulos anteriores, continuaremos a modelagem dos sistemas já iniciados, enfocando, desta vez, o diagrama de componentes. Como a maioria dos sistemas estudados são simples, os diagramas de componentes serão pequenos, alguns até bastante semelhantes entre si, devido a existirem poucos módulos de software, na maioria das vezes.

12.5.1 Sistema de Controle de Cinema

Desenvolva o diagrama de componentes para um sistema de controle de cinema sabendo que:

- É preciso existir um módulo para gerir a venda de ingresso aos clientes. Esse módulo deve gerar os ingressos e os emitir por meio da interface (que pode ser física também) para os clientes do cinema.
- O sistema necessita de um SGBD para persistir suas informações.
- Finalmente, existe a necessidade de um módulo de manutenção do sistema, onde basicamente serão mantidos os cadastros de sessões, salas, filmes, atores, gêneros etc.

12.5.2 Sistema de Controle de Clube Social

Desenvolva o diagrama de componentes referente ao sistema de clube social, levando em consideração os seguintes fatos:

- O sistema precisa de um módulo para manter o cadastro de seus sócios, dependentes e suas categorias.
- É preciso existir um SGBD para persistir esses dados.
- É necessário ainda haver um módulo para a emissão e quitação de mensalidades.

12.5.3 Sistema de Locação de Veículos

Desenvolva o diagrama de componentes para um sistema de aluguel de veículos, considerando que:

- O sistema precisa de um módulo para gerenciar o cadastro dos clientes, automóveis, modelos e marcas.
- É preciso existir um SGBD para gravar e recuperar os dados do sistema.
- É necessário ainda haver um módulo para realizar as locações dos automóveis da empresa, bem como registrar as devoluções dessas mesmas locações.

12.5.4 Sistema para Controle de Leilão Via Internet

Desenvolva o diagrama de componentes relativo a um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- O sistema necessita de uma página por meio da qual os participantes poderão se logar ou se autorregistrar. Essa página será útil também para apresentar os itens anunciados pelo leiloeiro e para receber as ofertas dos participantes, bem como para anunciar um item como arrematado.
- O sistema precisa de um módulo que gerencie o login dos participantes. É recomendável existir um módulo associado ao módulo de login para permitir o autorregistro das pessoas interessadas em participar do leilão.
- É necessário haver um módulo que permita registrar novos leilões, bem como cadastrar os itens que serão anunciados nos mesmos.
- Também é necessário um módulo responsável por gerenciar cada leilão, que permita a um leiloeiro abrir um leilão, anunciar os itens do leilão, receber lances, anunciar vencedores e arrematar itens.

- Finalmente, é necessário um sistema gerenciador de banco de dados para persistir e recuperar as informações necessárias ao sistema.

12.5.5 Sistema de Controle de Hotelaria

Desenvolva o diagrama de componentes para um sistema de controle de hotelaria, de acordo com as seguintes definições:

- Como de praxe, é necessário haver um módulo de manutenção para gerenciar os cadastros de funcionários, categorias, quartos etc.
- É necessário também implementar um módulo para realizar as reservas de quartos pelos hóspedes.
- Da mesma forma é preciso existir um módulo para gerenciar o aluguel dos quartos.
- O aluguel de um quarto pode implicar na solicitação de serviços ou no consumo de itens do frigobar, assim é necessário haver alguma forma de registrar esses pedidos.

12.6 Solução dos Exercícios

12.6.1 Sistema de Controle de Cinema

A figura 12.8 apresenta a solução para esse exercício.

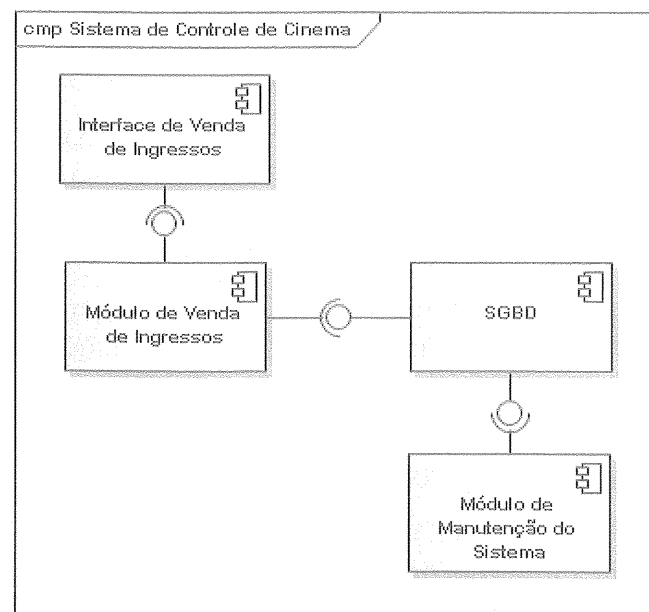


Figura 12.8 – Diagrama de Componentes – Sistema de Controle de Cinema.

Esse diagrama representa os seguintes componentes:

- **Interface de Venda de Ingressos**

Esse componente representa a interface de vendas de ingressos, abrangendo não somente a interface do sistema, mas também o hardware necessário para a impressão de ingressos.

- **Módulo de Venda de Ingressos**

Esse módulo é responsável por gerenciar o processo de venda de ingressos para o cinema.

- **SGBD**

Esse componente representa o sistema gerenciador de banco de dados responsável por manter as informações do sistema.

- **Módulo de Manutenção do Sistema**

Esse módulo é responsável pela manutenção dos cadastros do sistema, como filmes, sessões, salas, atores, gêneros etc.

12.6.2 Sistema de Controle de Clube Social

A solução para esse exercício é apresentada na figura 12.9.

Esse diagrama representa os seguintes componentes:

- **Módulo de Manutenção**

Esse módulo é responsável pela manutenção dos cadastros do sistema, como sócios, dependentes e categorias.

- **Módulo de Mensalidades**

Esse módulo é responsável por gerar as mensalidades devidas pelos sócios, bem como realizar as quitações das mesmas.

- **SGBD**

Esse componente representa o sistema gerenciador de banco de dados responsável por manter as informações do sistema. Observe que os dois módulos anteriores apresentam interfaces requeridas com esse componente, uma vez que solicitam operações de gravação e recuperação de dados no SGBD.

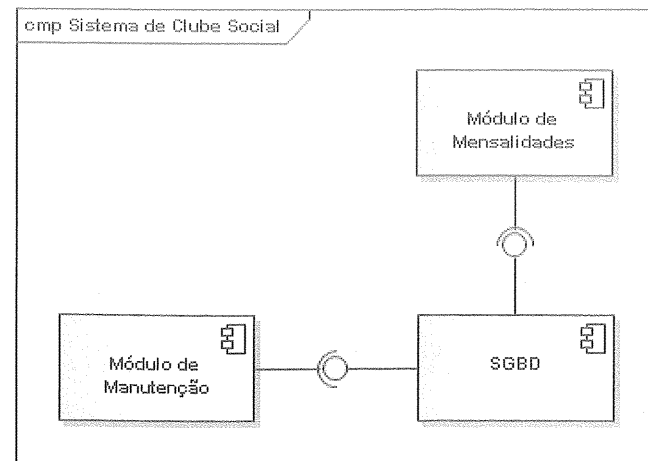


Figura 12.9 – Diagrama de Componentes – Sistema de Controle de Clube Social.

12.6.3 Sistema de Locação de Veículos

A solução desse exercício está contida na figura 12.10.

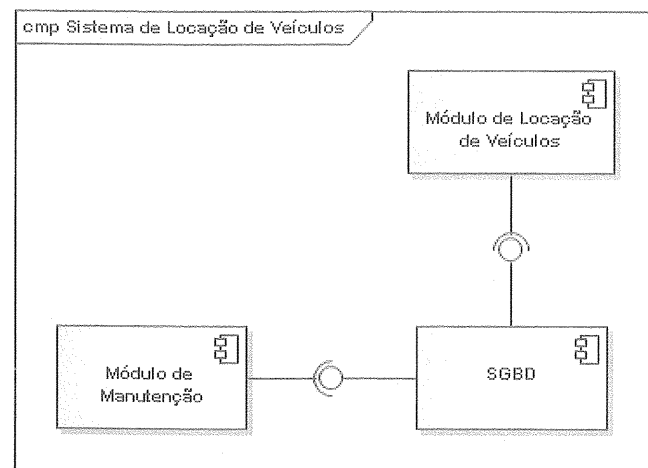


Figura 12.10 – Diagrama de Componentes – Sistema de Locação de Veículos.

Esse diagrama é muito semelhante ao apresentado na solução anterior. Por esse motivo explicaremos apenas o **Módulo de Locação de Veículos**, cuja função é gerenciar as locações de veículos da empresa, bem como as devoluções das mesmas.

12.6.4 Sistema para Controle de Leilão Via Internet

A seguir explicaremos os componentes do diagrama que representa a solução desse exercício, apresentado na figura 12.11.

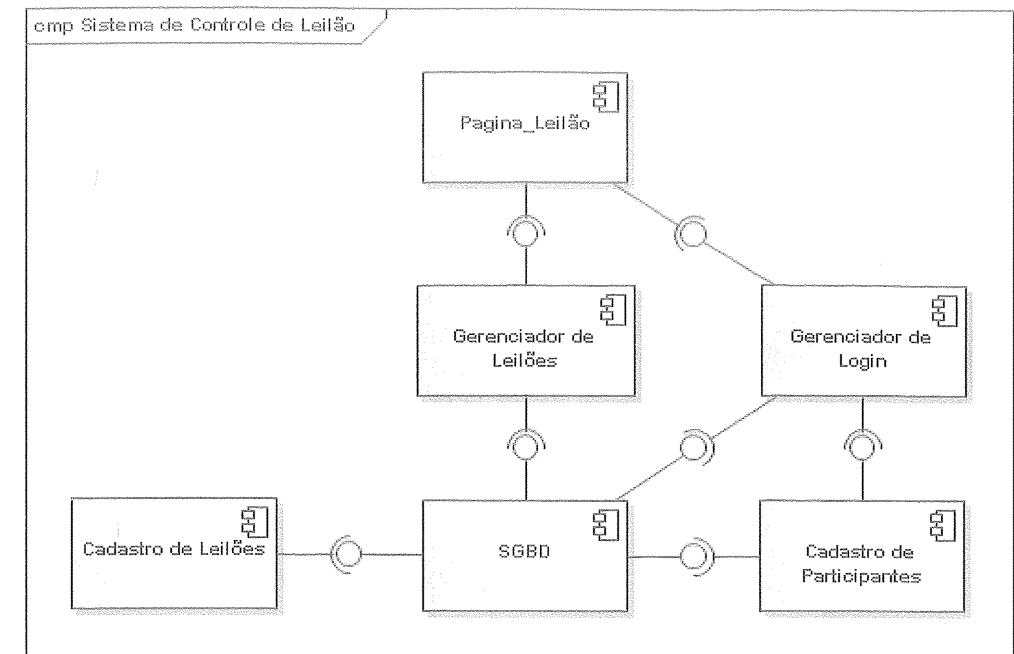


Figura 12.11 – Diagrama de Componentes – Sistema para Controle de Leilão Via Internet.

Esse diagrama representa os seguintes componentes:

■ **Página_Leilão**

Esse módulo é responsável pela manutenção dos cadastros do sistema, como sócios, dependentes e categorias. Observe que essa página tem interfaces requeridas com os módulos **Gerenciador de Login** e **Gerenciador de Leilões**, uma vez que são eles que realmente implementam os serviços oferecidos na página.

■ **Gerenciador de Login**

Esse módulo gerencia o login dos participantes no sistema, não permitindo que pessoas não registradas participem do leilão.

■ **Cadastro de Participantes**

Esse módulo está associado ao anterior, sendo responsável por permitir o autocadastro de novos participantes.

■ **Cadastro de Leilões**

Esse módulo permite cadastrar novos leilões, bem como os itens a eles associados.

■ Gerenciador de Leilões

Este é o módulo principal do sistema, por meio do qual um leilão é iniciado, seus itens são anunciados, os lances são recebidos, e os itens, arrematados.

■ SGBD

Como de praxe, esse componente representa o sistema gerenciador de banco de dados que mantém as informações do sistema.

12.6.5 Sistema de Controle de Hotelaria

Os componentes representados na figura 12.12, que representa a solução desse exercício, serão explicados a seguir.

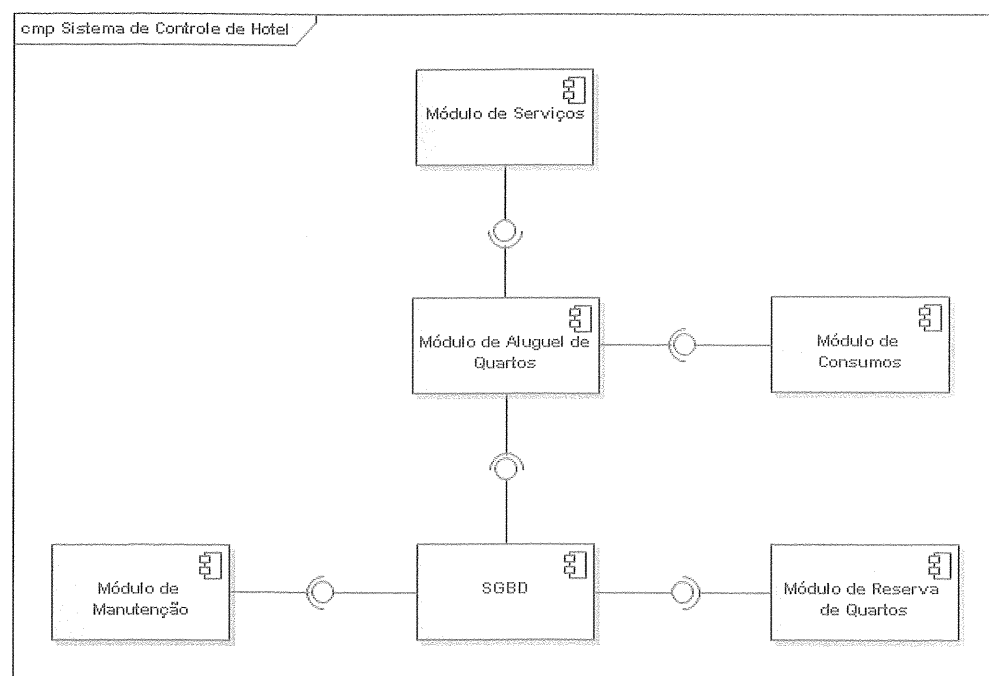


Figura 12.12 – Diagrama de Componentes – Sistema de Controle de Hotelaria.

Esse diagrama representa os seguintes componentes:

■ Módulo de Manutenção

Esse módulo é responsável pela manutenção dos cadastros do sistema, como funcionários, categorias, quartos, itens de frigobar, serviços oferecidos etc.

■ Módulo de Reserva de Quartos

Esse módulo tem como função permitir a um funcionário reservar um quarto para um hóspede.

■ Módulo de Aluguel de Quartos

Este é um dos módulos mais importantes do sistema e é responsável por permitir o aluguel de quartos por um funcionário do hotel para um hóspede. Observe que este módulo contém interfaces requeridas com os dois módulos seguintes, uma vez que esse módulo solicita o registro dos serviços pedidos por um hóspede e o registro do consumo de frigobar para esses dois módulos.

■ Módulo de Serviços

Esse módulo está associado ao módulo anterior e permite registro de pedido de serviços por um hóspede.

■ Módulo de Consumos

Esse módulo também está associado ao Módulo de Aluguel de Quartos, e sua função é registrar o consumo de frigobar por um hóspede.

■ SGBD

Esse módulo é responsável pelo armazenamento e recuperação das informações necessárias ao sistema.

CAPÍTULO 13

Diagrama de Implantação

O diagrama de implantação é o diagrama com a visão mais física da UML. Enfoca a questão da organização da arquitetura física sobre a qual o software será implantado e executado em termos de hardware, ou seja, as máquinas (computadores pessoais, servidores etc.) que suportarão o sistema, além de definir como essas máquinas estarão conectadas e por meio de quais protocolos se comunicarão e transmitirão informações. Esse diagrama também permite demonstrar como se dará a distribuição dos módulos do sistema, em situações em que estes sejam executados em mais de um servidor.

Um diagrama de implantação obviamente só é útil quando o sistema modelado for executado sobre múltiplas máquinas, as quais executem determinados módulos do software ou armazenem arquivos necessários ao mesmo. Se o software for projetado para ser executado sobre uma única máquina individual que não se comunique com outro hardware, não é necessário modelar um diagrama de implantação.

13.1 Nós

Nós são os componentes básicos de um diagrama de implantação. Um nó pode representar um item de hardware, como um servidor onde um ou mais módulos do software são executados ou que armazene arquivos consultados pelos módulos do sistema. Um nó pode representar também um ambiente de execução, ou seja, um ambiente que suporta o sistema de alguma forma. Nós podem conter outros nós, sendo possível encontrar um nó que representa um item de hardware contendo outro nó que representa um ambiente de execução, embora um nó que represente um item de hardware possa conter outros nós representando outros itens de hardware e um nó que represente um ambiente de execução possa conter outros ambientes de execução.

Quando um nó representa um hardware, este pode apresentar o estereótipo `<<device>>`, já quando um nó representa um ambiente de execução pode utilizar o estereótipo `<<ExecutionEnvironment>>`. Exemplos de ambientes de execução são

sistemas operacionais ou sistemas de banco de dados. Um nó é representado por um cubo contendo um texto indicando o nome do item de hardware ou ambiente de execução que ele representa. A figura 13.1 apresenta um exemplo de nó, representando um item de hardware.

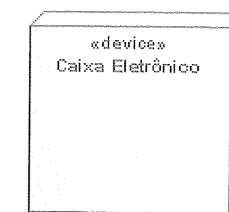


Figura 13.1 – Exemplo de Nó.

Essa figura apresenta um exemplo de nó que se refere a um caixa eletrônico qualquer, sem especificar um em especial. Podemos, no entanto, definir um nome específico para a máquina que o nó representa, como é comum encontrar nas organizações que têm grandes redes com muitos servidores espalhados, conforme demonstra a figura 13.2.

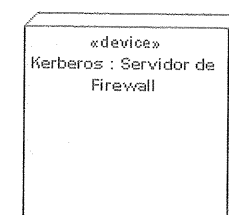


Figura 13.2 – Nó com Denominação.

Nesse exemplo, determinamos que o nó representa um servidor de firewall e que o nome pelo qual esse servidor é conhecido na organização é Kerberos. Um nó pode ainda conter detalhes de configuração do hardware que ele representa. Os detalhes de configuração podem ser definidos por meio de etiquetas (tags), que são uma forma de estabelecer propriedades extras para um elemento. A figura 13.3 mostra um exemplo de nó com sua configuração detalhada.

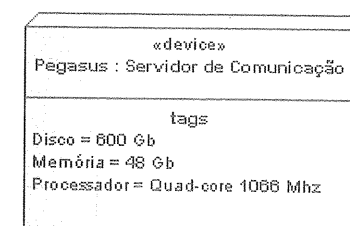


Figura 13.3 – Nó com Detalhes de Configuração.

Aqui utilizamos outro nó, que representa um servidor de comunicação denominado Pegasus. A configuração desse servidor é definida como tendo 600 gigabytes de disco, 48 gigabytes de memória RAM e um processador quad-core com velocidade de 1066 megahertz.

13.2 Estereótipos

Existem outros estereótipos que podem ser aplicados a nós, além do estereótipo genérico <<device>>, que podem ser aplicados a itens de hardware, tais como <<computer>>, que representa um computador mais simples; <<secure>>, que representa um hardware de segurança responsável por impedir invasões à rede interna da organização; <<server>>, que representa um servidor; e <<storage>>, que representa um hardware cuja função é armazenar informações, como um servidor de banco de dados ou de arquivos. A figura 13.4 apresenta exemplos desses quatro estereótipos (existem outros), na ordem que foram citados, da esquerda para direita e de cima para baixo. Assim, o nó Caixa Eletrônico representa um computador simples, o nó Kerberos identifica um hardware de segurança (no caso um firewall), o nó Poseidon representa um servidor de aplicação e o nó Hera um servidor de banco de dados. Observe que esses estereótipos são todos gráficos, uma vez que modificam de alguma forma o desenho-padrão do elemento.



Figura 13.4 – Nós com Estereótipos.

13.3 Associações

Os nós podem ter ligações físicas entre si de forma que possam se comunicar e trocar informações. Tais ligações denominam-se associações e são representadas por linhas ligando um nó a outro. A figura 13.5 apresenta um exemplo de associação entre nós.

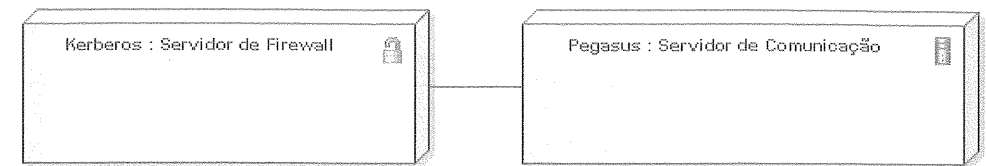


Figura 13.5 – Associação entre Nós.

Como podemos observar nessa figura, o servidor de firewall Kerberos está associado ao servidor de comunicação Pegasus. As associações, além de determinarem uma ligação física entre os nós, podem também determinar a forma como eles se comunicam, ou seja, o protocolo de comunicação utilizado, pelo uso de estereótipos. Um exemplo de nós associados utilizando estereótipos é apresentado na figura 13.6.

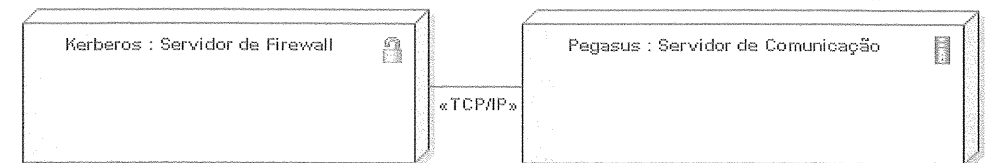


Figura 13.6 – Associação entre Nós com Estereótipo.

Ao observarmos essa figura percebemos que os nós Kerberos e Pegasus estão associados e comunicam-se por meio do protocolo TCP/IP.

13.4 Exemplo de Diagrama de Implantação

O exemplo apresentado na figura 13.7 enfoca o ambiente físico onde será executado o sistema de controle bancário que vem sendo desenvolvido ao longo de diversos capítulos deste livro.

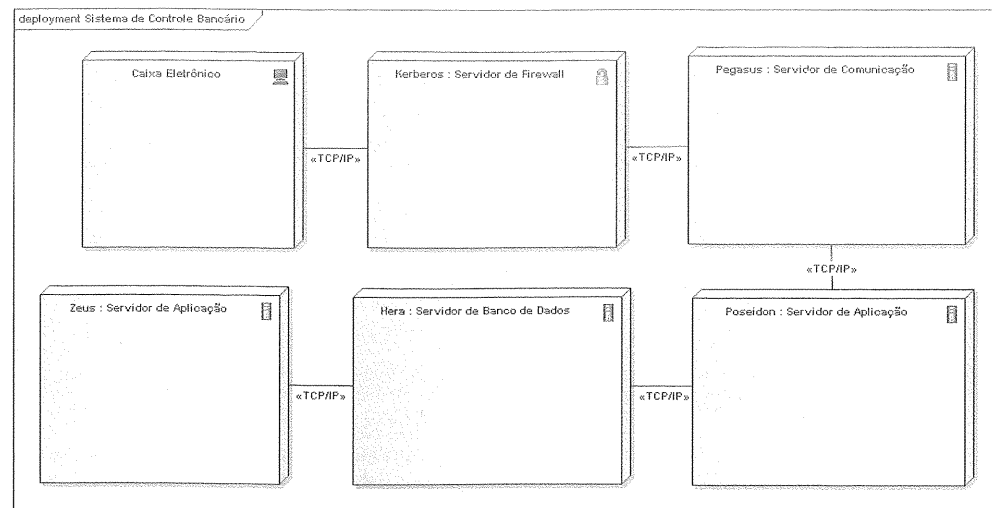


Figura 13.7 – Diagrama de Implantação – Sistema de Controle Bancário.

Nesse exemplo, existe um grande número de terminais de caixa eletrônico, muitos deles distantes geograficamente, em conexão com o sistema bancário. Obviamente não há como detalhar cada um deles. Assim, o nó Caixa Eletrônico (cujo estereótipo indica que ele é um computador simples), representa todos os terminais de caixa eletrônico disponibilizados pela instituição bancária.

O nó Caixa Eletrônico comunica-se com o servidor de firewall Kerberos, responsável por impedir a invasão de pessoas não-autorizadas na empresa, com demonstra seu estereótipo. O servidor Kerberos repassa as mensagens recebidas (se estas forem consideradas válidas e seguras) para o servidor de comunicação Pegasus. Esse servidor é necessário, uma vez que o sistema deve ser capaz de suportar uma quantidade muito grande de conexões, que precisarão ser gerenciadas.

O servidor Pegasus repassa as mensagens para o servidor de aplicação Poseidon, onde o módulo de gerenciamento de autoatendimento está sendo executado, sendo este responsável por interpretar os eventos ocorridos no caixa eletrônico. O servidor Poseidon, por sua vez, comunica-se com o servidor de banco de dados Hera, que é um servidor de armazenamento, como se pode notar por seu estereótipo, no qual estará executando algum tipo de sistema gerenciador de banco de dados para armazenar e recuperar as informações necessárias ao sistema. Observe que existe ainda um segundo servidor de aplicação, chamado Zeus, onde são executados outros módulos do sistema de controle bancário, como o gerenciador de contas.

13.5 Artefatos

Um artefato é uma entidade física, um elemento concreto que existe realmente no mundo real, assim como os nós que o suportam. Um artefato pode ser um arquivo fonte, um arquivo executável, um arquivo de ajuda, um documento de texto etc. Um artefato deve estar implementado em um nó. A figura 13.8 apresenta um exemplo de artefato implementado em um nó.

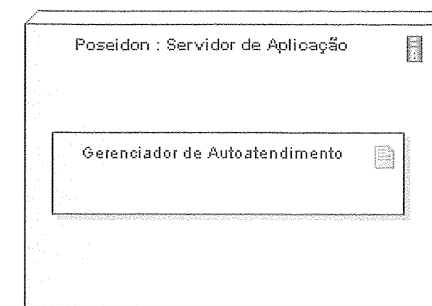


Figura 13.8 – Artefato implementado em um Nó.

O leitor notará que o artefato denominado Gerenciador de Autoatendimento tem a mesma denominação que um dos componentes apresentados no capítulo sobre o diagrama de componentes. Na verdade, um artefato é muitas vezes uma “manifestação” no mundo real de um componente. No entanto, não necessariamente existirá um artefato para cada componente, sendo possível existirem diversos artefatos manifestados a partir de um único componente. A figura 13.9 apresenta um exemplo de artefato instanciado a partir de um componente. Observe que existe um relacionamento de dependência entre o componente e o artefato, contendo o estereótipo <<manifest>>, significando que o artefato é uma representação do componente no mundo real.

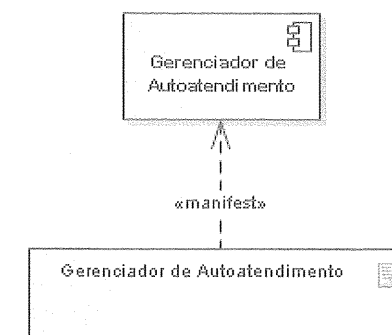


Figura 13.9 – Artefato manifestado a partir de um Componente.

Pode-se perceber, ao examinarmos essa figura, que o artefato Gerenciador de Autoatendimento é uma manifestação do componente de mesmo nome.

Outra forma de demonstrar que um artefato está contido em um nó é também por meio de um relacionamento de dependência, contendo o estereótipo <<deploy>>, entre o nó e os artefatos, conforme demonstra a figura 13.10.

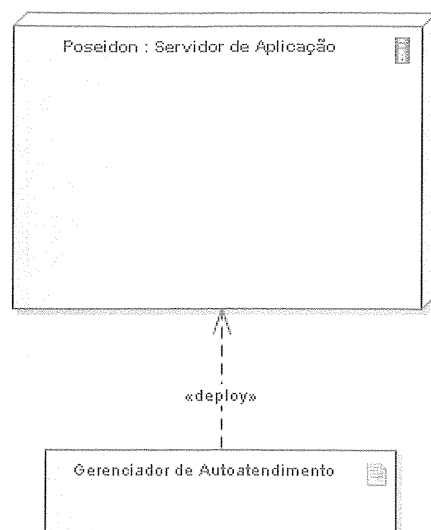


Figura 13.10 – Artefato implementado em um Nó.

13.6 Especificação de Implantação

Especifica um conjunto de propriedades que determinam parâmetros de execução de um artefato implementado em um nó, conforme demonstra a figura 13.11.

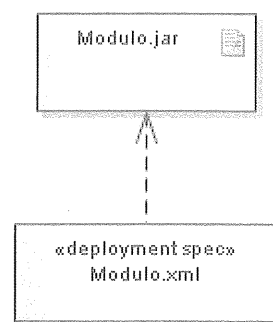


Figura 13.11 – Especificação de Implantação.

13.7 Exemplo de Diagrama de Implantação contendo Artefatos

Nesta seção ilustraremos um diagrama de implantação contendo artefatos, como pode ser visto na figura 13.12.

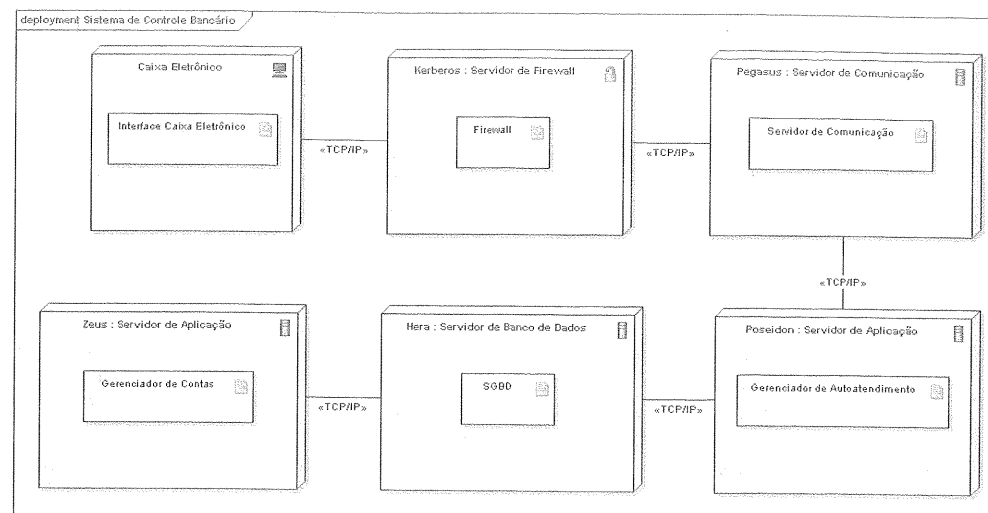


Figura 13.12 – Diagrama de Implantação Contendo Artefatos.

Ao examinar essa figura, o leitor perceberá que tomamos o primeiro exemplo de diagrama de implantação e o complementamos com artefatos. Esses artefatos são manifestações dos componentes apresentados no diagrama de componentes do sistema de controle bancário estudado no capítulo anterior.

Dessa forma, o nó Caixa Eletrônico suporta o módulo denominado Interface Caixa Eletrônico, cuja função é repassar os eventos ocorridos sobre a interface para o sistema de controle bancário. Do mesmo modo, o nó Kerberos suporta um software de firewall, responsável por impedir que pessoas não-autorizadas invadam o sistema. Seguindo a mesma lógica, o nó Pegasus suporta um software responsável por estabelecer a comunicação externa do sistema.

O servidor Poseidon é, por sua vez, responsável por suportar o módulo de software Gerenciador de Autoatendimento, e o servidor Zeus por suportar o módulo de software Gerenciador de Contas. Finalmente, o nó Hera suporta um sistema gerenciador de banco de dados que armazena e recupera as informações necessárias ao sistema de controle bancário.

Aqui só representamos a ligação física entre os nós, mas os artefatos podem ter ligações lógicas entre si, como conexões de banco de dados, por exemplo.

13.8 Nós Contendo Pacotes

Um nó pode suportar subsistemas ou mesmo sistemas inteiros. Esses sistemas podem ser representados como pacotes, como demonstra a figura 13.13.

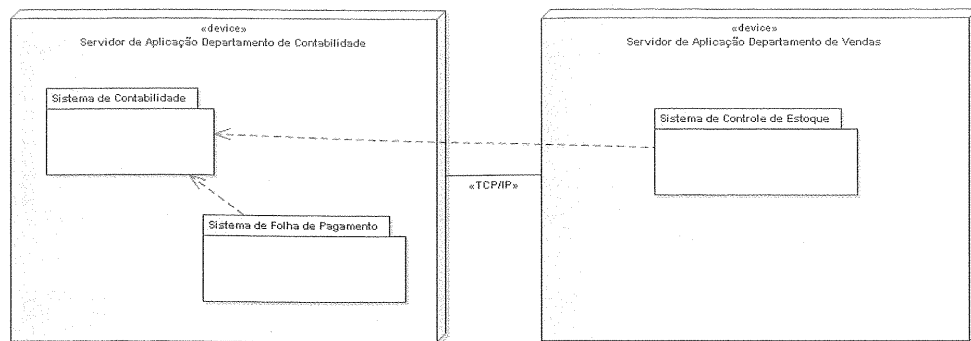


Figura 13.13 – Diagrama de Implantação Contendo Pacotes.

Nesse exemplo existem dois nós, o **Servidor de Aplicação Departamento de Contabilidade** e o **Servidor de Aplicação Departamento de Vendas**. O primeiro suporta dois sistemas representados pelos pacotes **Sistema de Contabilidade** e **Sistema de Folha de Pagamento**, enquanto o segundo suporta o sistema representado pelo pacote **Sistema de Controle de Estoque**. Uma vez que esses sistemas estão integrados, existe um relacionamento de dependência entre eles.

13.9 Exercícios Propostos

Aqui será finalizada a modelagem dos sistemas que temos exercitado desde o capítulo referente ao diagrama de casos de uso. Porém, a maioria dos sistemas que temos modelado, por se tratarem de sistemas relativamente simples, não necessita de um grande número de servidores, portanto, consideramos válido exercitar somente o sistema de controle de leilão.

13.9.1 Sistema para Controle de Leilão Via Internet

Desenvolva o diagrama de implantação para o sistema de controle de leilão via internet, sabendo que:

- Os participantes acessam o sistema de leilão de suas próprias máquinas pessoais, de forma que é necessário representá-las.
- Uma vez que esse sistema é acessado externamente é necessário estabelecer uma linha de segurança para impedir invasões.

- O sistema precisa suportar uma quantidade potencialmente grande de conexões. Assim, é necessário existir um servidor de comunicação.
- Toda a aplicação pode rodar em um único servidor, não havendo necessidade de dividi-la.
- No entanto, deve haver um servidor de banco de dados para guardar fisicamente as informações do sistema, bem como recuperá-las quando solicitado. Embora este pudesse estar instalado no mesmo hardware, poderia ser útil se fosse executado em outra máquina.

13.10 Solução dos Exercícios

13.10.1 Sistema para Controle de Leilão Via Internet

A figura 13.14 apresenta a solução para esse exercício. Nessa solução já representamos os artefatos que manifestam os componentes relativos ao mesmo sistema definidos no capítulo 12.

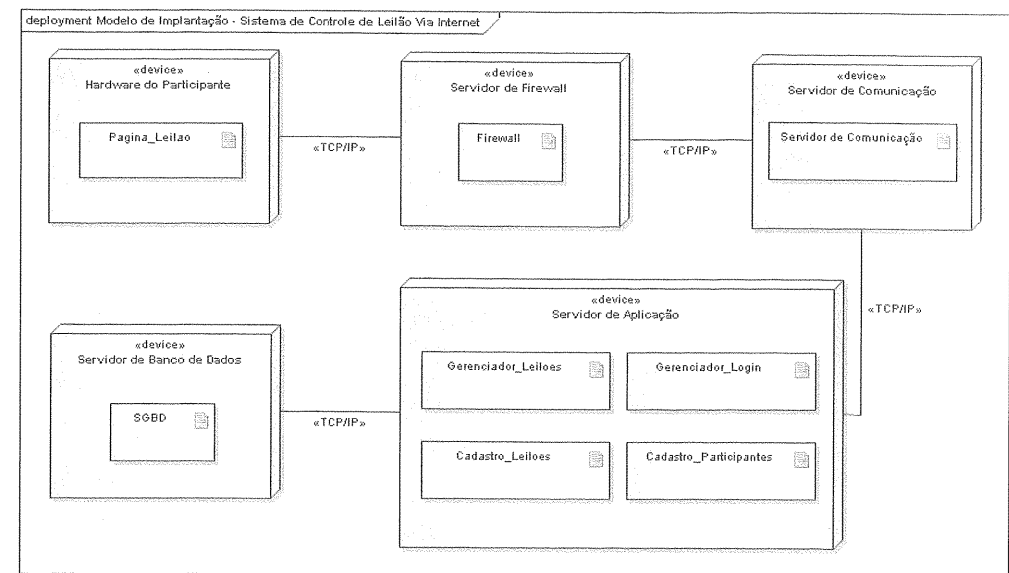


Figura 13.14 – Diagrama de Implantação – Sistema para Controle de Leilão Via Internet.

Esse diagrama representa os seguintes nós:

■ Hardware do Participante

Esse nó representa as máquinas das pessoas que se registraram para participar do leilão. O nó suporta o artefato que representa a página do leilão.

Essa página será atualizada sempre que um novo item for anunciado ou que um item receba um lance. Por meio dela um participante pode fazer ofertas para um item, se assim o desejar.

■ Servidor de Firewall

Esse nó representa o hardware que suporta o sistema de segurança da empresa. Nele é executado o módulo denominado **Firewall**, que, como seu nome dá a entender, representa um software de firewall, responsável por impedir invasões de usuários não-autorizados.

■ Servidor de Comunicação

Levando-se em consideração a possibilidade de existir um grande número de conexões externas relativas aos participantes de um leilão, considerou-se válido representar esse nó, que suporta um artefato representando um software de comunicação.

■ Servidor de Aplicação

Esse nó é responsável por suportar o sistema de leilão propriamente dito. Nele são executados os módulos **Gerenciador_Login**, **Cadastro_Participantes**, **Gerenciador_Leiloes** e **Cadastro_Leiloes**, que são manifestações dos componentes desse sistema, já explicados no capítulo anterior.

■ Servidor de Banco de Dados

Finalmente, esse nó suporta um sistema gerenciador de banco de dados responsável pela gravação e recuperação das informações do sistema.



CAPÍTULO 14

Diagrama de Estrutura Composta

Este é um dos três novos diagramas que passaram a existir a partir do lançamento da UML 2. O diagrama de estrutura composta pode ser utilizado para modelar colaborações. Uma colaboração descreve uma visão de um conjunto de entidades cooperativas interpretadas por instâncias que cooperam entre si para executar uma função específica. O termo estrutura desse diagrama refere-se a uma composição de elementos interconectados, representando instâncias de tempo de execução colaborando por meio de vínculos de comunicação para atingir algum objetivo comum. Esse diagrama também pode ser utilizado para descrever a estrutura interna de um classificador.

O diagrama de estrutura composta é semelhante ao diagrama de classes, porém, esse último apresenta uma visão estática da estrutura de classes, enquanto o primeiro tenta expressar arquiteturas de tempo de execução, padrões de uso e os relacionamentos dos elementos participantes, o que nem sempre pode ser representado por diagramas estáticos.

14.1 Colaborações

Uma colaboração é representada como um tipo de classificador e define um conjunto de entidades cooperativas que serão interpretadas por instâncias, que representarão o papel das entidades, bem como um conjunto de conectores que definem caminhos de comunicação entre as instâncias participantes. As entidades cooperativas são as propriedades da colaboração.

Colaborações são geralmente utilizadas para explicar como um conjunto de instâncias cooperando entre si realiza uma tarefa conjunta ou um conjunto de tarefas. O propósito primário de uma colaboração é explicar como um sistema

funciona, portanto, apresenta somente os aspectos extremamente necessários à explicação em questão, suprimindo quaisquer outros detalhes. Um mesmo objeto pode estar interpretando simultaneamente diversos papéis em diferentes colaborações, mas cada colaboração deverá representar somente os aspectos do objeto relevantes ao seu propósito. Uma colaboração é representada por uma elipse tracejada contendo uma descrição, conforme pode ser observado na figura 14.1.

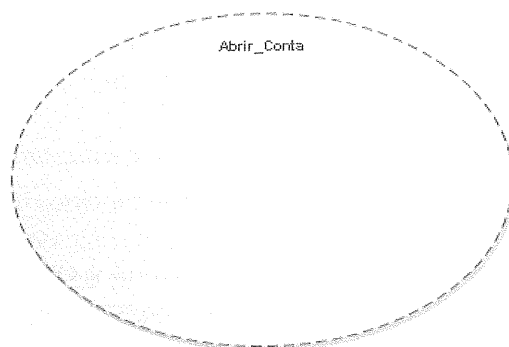


Figura 14.1 – Exemplo de Colaboração – Abertura de Conta.

Essa colaboração representa a tarefa de abertura de conta do sistema de controle bancário que temos modelado no decorrer do livro.

14.2 Papéis

Os “papéis” de uma colaboração são interpretados por instâncias que cooperam entre si para concluir uma tarefa. Os relacionamentos entre as instâncias considerados relevantes para a tarefa em questão são representados por meio da utilização de linhas ligando uma instância a outra, chamadas de conectores.

Os papéis de colaborações definem um uso de instâncias, enquanto os classificadores que definem esses papéis especificam todas as propriedades requeridas dessas instâncias. Assim, uma colaboração especifica quais propriedades uma instância deve ter habilitadas. Nem todas as características, nem todo o conteúdo das instâncias participantes nem todas as ligações dessas instâncias são sempre requeridas em uma colaboração particular, motivo pelo qual uma colaboração é muitas vezes descrita em termos de papéis definidos por interfaces, uma vez que estes são descrições de um conjunto de propriedades (características observáveis externamente) fornecidas ou requeridas por uma instância. A figura 14.2 demonstra um exemplo de colaboração com sua estrutura interna, ou seja, papéis e conectores.

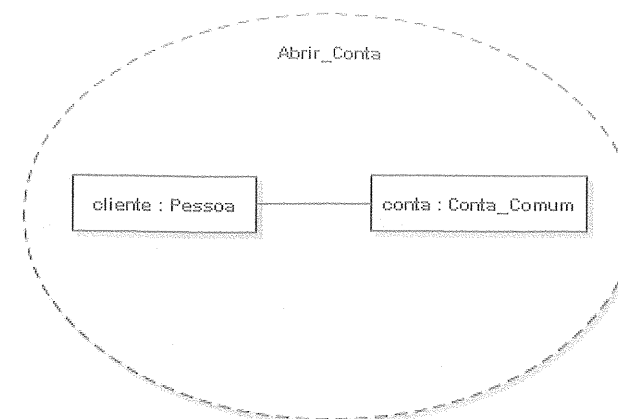


Figura 14.2 – Colaboração Contendo Papéis e Conectores.

Nesse exemplo, uma instância da classe Pessoa interpretando o papel de cliente colabora (interage) com uma instância da classe Conta_Comum, que interpreta o papel de conta. O objetivo dessa interação é permitir que uma nova conta possa ser aberta. A definição do classificador da instância a que pertence o papel não é obrigatória, mas às vezes facilita a compreensão da colaboração.

Uma colaboração pode conter outras colaborações dentro de si, conforme é demonstrado pela figura 14.3.

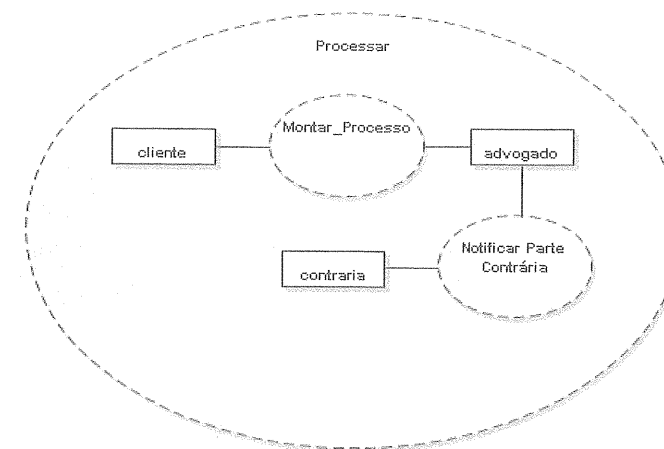


Figura 14.3 – Colaboração Contendo Colaborações.

Nesse exemplo, a colaboração Processar contém duas colaborações internas, as colaborações Montar_Processo e Notificar_Parte_Contrária. Na primeira, um cliente colabora com um advogado para montar um processo, e na segunda um advogado interage com uma parte contrária para notificá-la do processo.

14.3 Ocorrência de Colaboração

Uma ocorrência de colaboração representa a aplicação do padrão descrito por uma colaboração a uma situação específica envolvendo classes ou instâncias executando papéis específicos da colaboração. A denominação de uma ocorrência de colaboração apresenta a mesma notação utilizada na denominação de um objeto (uma vez que uma ocorrência de colaboração é uma instância de uma colaboração), ou seja, o nome da ocorrência seguida de dois pontos (:) e o nome da colaboração ou, caso não se queira dar um nome para a ocorrência, simplesmente dois pontos (:) e o nome da colaboração. Uma ocorrência de colaboração pode ser relacionada a uma classe por meio de uma associação de dependência contendo o estereótipo <<represents>>, conforme demonstra a figura 14.4.

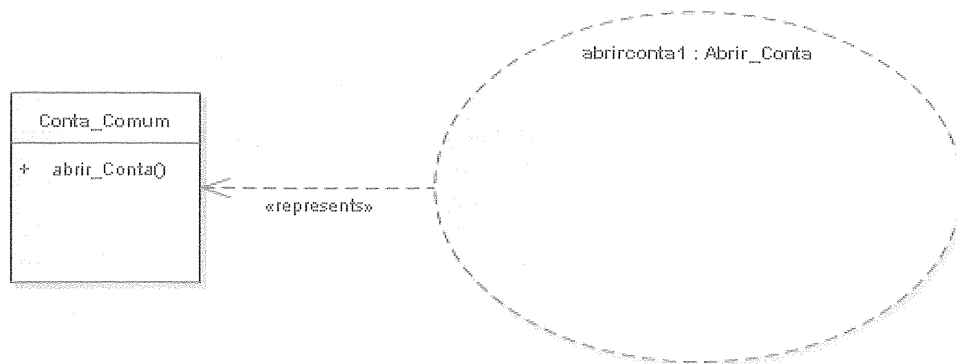


Figura 14.4 – Ocorrência de Colaboração Relacionada a uma Classe.

Esse exemplo determina que a ocorrência de colaboração *abrirconta1* está relacionada a um classificador, nesse caso, a classe *Conta_Comum*. Essa colaboração representa a atividade de abertura de conta, na qual o método *abrir_Conta* definido na classe *Conta_Comum* desempenha uma função muito importante.

Pode-se também representar uma ocorrência de colaboração a partir de uma colaboração por meio do mesmo relacionamento de dependência, dessa vez contendo o estereótipo <<occurrence>>, conforme demonstra a figura 14.5.

Nesse exemplo percebemos que *Abrir_Conta_Especial* é uma ocorrência de colaboração da colaboração *Abrir_Conta* e que a instância *conta_especial* da ocorrência *abrircontesp1* interpreta o papel de *conta* do elemento da colaboração *Abrir_Conta*.

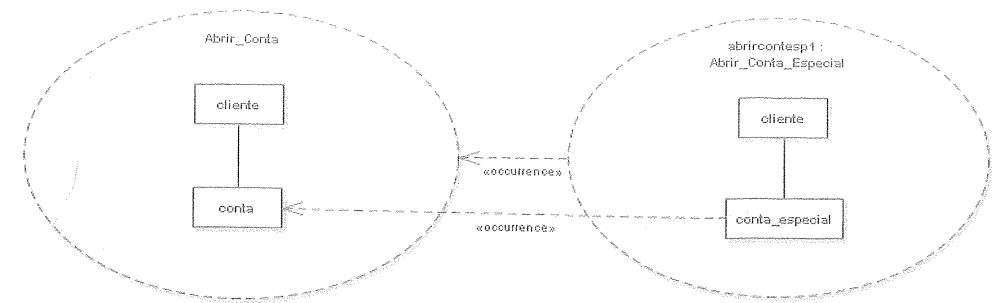


Figura 14.5 – Ocorrência de Colaboração.

14.4 Portas

Como já foi introduzido no diagrama de classes, portas são características estruturais de um classificador que representam pontos de interação distintos entre um classificador e seu ambiente ou entre o classificador e suas partes internas. Portas são conectadas às propriedades de um classificador por meio de conectores, por meio dos quais requisições podem ser feitas para invocar as características comportamentais do classificador. Uma porta é utilizada para representar os serviços que um classificador fornece a seu ambiente ou os serviços que requer deste. Portas são representadas por quadrados sobrepostos à borda de um classificador ou de suas partes internas. Uma porta pode ter um nome ou não. A figura 14.6 apresenta um exemplo de porta.

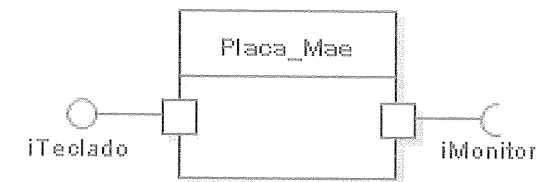


Figura 14.6 – Exemplo de Portas.

Nesse exemplo, a classe *Placa_Mae* tem duas portas, uma para cada interface com seu ambiente externo. A classe está totalmente encapsulada, ou seja, não possuímos nenhum conhecimento a respeito de sua estrutura interna.

14.5 Propriedades e Partes

Uma propriedade representa um conjunto de instâncias internas, que são possuídas por uma instância de um classificador contêiner. Quando uma instância de um classificador é criada, um conjunto de instâncias correspondente às suas propriedades pode ser criado também. Uma propriedade específica que um conjunto de instâncias pode existir. Esse conjunto é um subconjunto do conjunto total de instâncias especificado pelo classificador que define a propriedade.

Uma parte declara que uma instância desse classificador pode conter um conjunto de instâncias por composição, ou seja, essas instâncias não podem relacionar-se com outro objeto, pois pertencem exclusivamente à instância da classe contêiner e serão destruídas quando essa instância for destruída. A figura 14.7 apresenta um exemplo de partes.

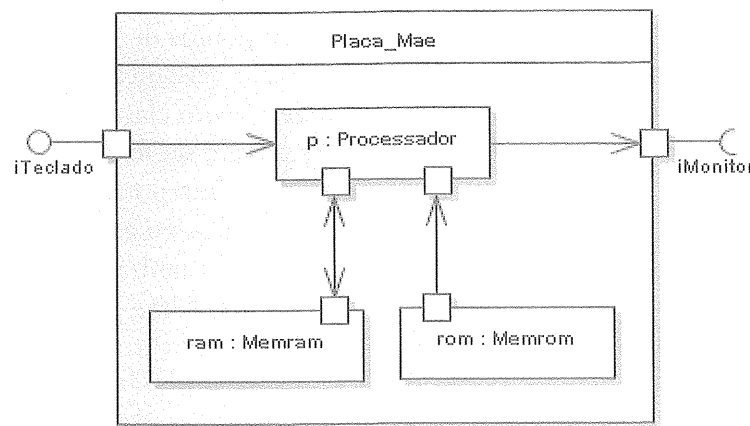


Figura 14.7 – Exemplo de Partes.

Nesse exemplo apresentamos a estrutura interna de uma placa-mãe, composta nesse exemplo por partes que representam processador, memória RAM e memória ROM. As linhas que ligam as partes entre si e ao ambiente externo por meio das portas são chamadas conectores e podem ser unidirecionais, se a seta aponta somente em uma direção, ou bidirecionais, se apontam para as duas direções, significando que podem receber ou enviar informações ou ambos. O conector que liga o processador à memória ROM é unidirecional, porque a memória é somente leitura, assim a informação é unidirecional.

Uma propriedade que especifique uma instância que não pertença por composição à instância do classificador container é apresentada como um retângulo tracejado e dita referenciada. Partes podem conter a definição de sua multiplicidade, podendo esta ser fixa, quando se sabe o número exato de

instâncias contidas na instância do classificador contêiner, ou determinando o número mínimo e máximo da multiplicidade, conforme demonstra o exemplo da figura 14.8.

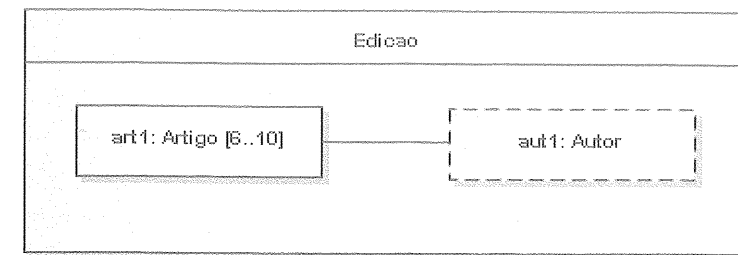


Figura 14.8 – Exemplo de Parte com Multiplicidade e Propriedade Referenciada.

Aqui tomamos o exemplo da figura 4.10 do capítulo sobre o diagrama de classes, onde uma edição relaciona-se por composição a diversos artigos, significando que uma instância da classe Artigo relaciona-se única e exclusivamente com uma instância da classe Edicao. Como se pode observar a parte que representa os artigos da edição apresenta uma multiplicidade que determina que devem existir no mínimo seis instâncias da classe artigo dentro da instância da classe Edicao e no máximo dez. Ao mesmo tempo, existe uma propriedade referenciada, como podemos observar pelo tracejado de seu retângulo, uma vez que instâncias da classe Autor devem estar associadas a um artigo, mas não se relacionam à edição por composição.

CAPÍTULO 15

Diagrama de Tempo ou de Temporização

Esse diagrama apresenta algumas semelhanças com o diagrama de máquina de estados. No entanto, ele enfoca as mudanças de estado de um objeto ao longo do tempo. Esse diagrama terá pouca utilidade para modelar aplicações comerciais, contudo, poderá ser utilizado na modelagem de sistemas de tempo real ou sistemas que utilizem recursos de multimídia/hipermídia, onde o tempo em que um objeto executa algo é muitas vezes importante. A figura 15.1 apresenta um exemplo de diagrama de tempo.

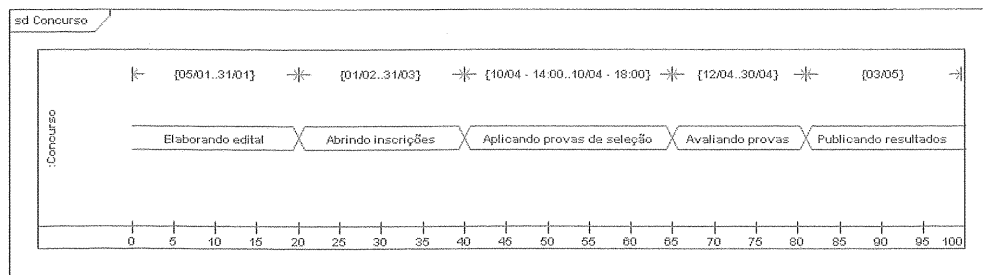


Figura 15.1 – Diagrama de Tempo – Linha de Vida de Valor.

É importante destacar que o diagrama de tempo tem duas notações ou formas de representação: uma notação conhecida como concisa, mais simples, chamada linha de vida de valor, adotada no exemplo da figura 15.1, e uma notação considerada mais robusta, onde as etapas são apresentadas em uma forma semelhante a um gráfico, chamada linha de vida de estado. No diagrama de tempo, o termo linha de vida (lifeline) refere-se ao caminho percorrido por um objeto durante um determinado tempo.

No exemplo da figura 15.1 utilizamos a forma concisa, ou seja, a linha de vida de valor. Nesse exemplo descrevemos as etapas pelas quais passa um concurso (o objeto da linha de vida), desde a elaboração de seu edital até a apresentação de seus resultados. Cada etapa ou estado do objeto da classe Concurso é apresentada por meio de um hexágono, sendo que o primeiro e último estados se encontram abertos. Acima de cada etapa, entre barras verticais, se encontram as restrições de duração que determinam o tempo em que transcorrem as etapas. No caso do estado *Abrindo Inscrições*, o período vai de 1º de fevereiro a 31 de março.

A figura 15.2 apresenta o mesmo diagrama, dessa vez utilizando sua forma robusta, linha de vida de estado, onde as transições de estado são determinadas por mudanças em um gráfico, podendo estas conter descrições que determinam o evento que causou a mudança, se isso for considerado necessário.

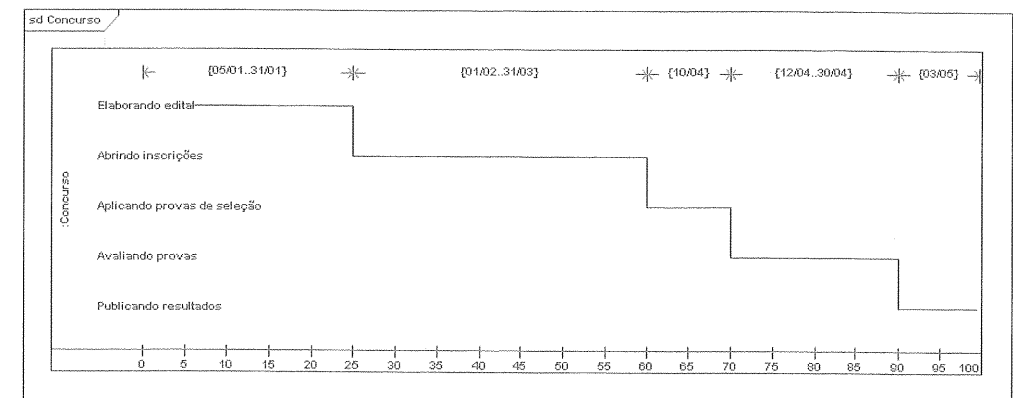


Figura 15.2 – Diagrama de Tempo – Linha de Vida de Estado.

Um diagrama de tempo pode ter linhas de vida de múltiplos objetos, utilizando a mesma notação ou notações diferentes.

CAPÍTULO 16

Estudo de Caso – Sistema de Pizzaria Online – PizzaNet

Neste capítulo modelaremos um sistema de pizzaria online fictício, cujos pedidos poderão ser feitos pela internet. Deve ficar claro, porém, que os diagramas apresentados ao longo deste capítulo não estão atrelados a nenhuma linguagem de programação específica, podendo ocorrer que alguns dos processos aqui modelados venham a ser simplificados, ou mesmo que alguns métodos possam ser considerados desnecessários, sendo possível serem incluídos dentro das instruções de outros métodos. Nosso objetivo foi produzir diagramas de fácil compreensão que possam ser entendidos por qualquer leitor, sem que o mesmo tenha que ter conhecimento a respeito de uma determinada linguagem de programação para entendê-los.

O sistema será modelado por meio de quase todos os diagramas da UML, excetuando-se o diagrama de estrutura composta, que consideramos desnecessário modelar, e o diagrama de tempo, que não se aplica a essa situação.

O leitor que desejar exercitar seus conhecimentos poderá tentar modelar o sistema a partir do enunciado do problema proposto, descrito na próxima seção, e depois corrigi-lo com base na solução apresentada ao longo deste capítulo.

16.1 Descrição do Problema

Uma empresa necessita de um sistema de pizzaria online, por meio do qual seus clientes possam solicitar pizzas pela Internet. Para modelar esse sistema devemos levar em consideração os seguintes requisitos apresentados pela empresa:

- A tela inicial do site da pizzaria deve apresentar duas divisões verticais. Na divisão da esquerda devem aparecer os ícones intitulados **Logar**, **Pizzas**, **Bebidas**, **Visualizar Pedido**, **Sabores Mais Pedidos**, **Pedidos Anteriores**, **Concluir Pedido** e **Opinar**.
- Já a divisão da direita carregará o módulo escolhido pelo usuário quando este selecionar um dos botões já descritos. Por padrão, deverá ser carregado o módulo para escolha de pizzas, que poderá ser chamado também por meio do botão **Pizzas**. Nesse módulo a tela deverá ser subdividida em duas divisões horizontais, onde, na superior, o usuário poderá escolher entre três ícones que representam os tamanhos que podem ser escolhidos para a pizza (pequena – 4 pedaços, média – 6 pedaços, grande – 8 pedaços). Ao selecionar o tamanho da pizza, o sistema permite que o cliente escolha, na segunda divisão horizontal, tantos sabores quantos os permitidos pelo tamanho da pizza (1, 2 ou 4). Essa divisão contém ainda um ícone para adicionar a pizza escolhida ao pedido, permitindo que o cliente escolha outras pizzas após isso. Assim, ao terminar de escolher os tamanhos da pizza, basta clicar no botão **Adicionar**, o que adicionará a pizza ao pedido e permitirá também a escolha de outra pizza, se o cliente assim o desejar. O valor da pizza é calculado pelo preço do sabor mais caro multiplicado pelo número de pedaços da pizza.
- O botão **Logar** deverá permitir que o cliente se autentique no sistema, o que é necessário para que este possa concluir o pedido. Esse módulo solicita ao cliente que informe seu **nome-login** e sua senha para logá-lo. Caso o cliente não esteja cadastrado, esse módulo deverá permitir que o cliente solicite a execução do módulo **Autorregistrar**, onde poderá se cadastrar.
- O botão **Bebidas** deverá apresentar um formulário um pouco semelhante ao módulo de escolha de pizzas, apresentando também uma divisão horizontal onde, na parte superior, deverão ser apresentados os tipos de bebidas oferecidas (suco, refrigerante e cerveja) e, após a escolha do tipo desejado, o sistema deverá apresentar na segunda divisão todas as bebidas do tipo escolhido disponíveis, permitindo ao usuário selecionar e adicionar ao pedido quantas bebidas quiser.
- O botão **Visualizar Pedido** deverá apresentar todos os itens escolhidos pelo cliente (pizzas e bebidas) até o momento, bem como o valor total do pedido, permitindo que o cliente exclua algum item, se assim o desejar.

- O botão **Fechar Pedido** deverá permitir que o cliente conclua o pedido. Nesse processo, o cliente deverá obrigatoriamente se logar, caso ainda não o tenha feito, podendo alterar seus dados, se desejar, ou se cadastrar no sistema, caso esta seja a primeira vez em que o cliente realiza um pedido na PizzaNet. Após essa verificação, o sistema deverá executar o módulo **Visualizar Pedido** para apresentar os itens escolhidos pelo cliente. Em seguida o sistema deverá apresentar ainda o endereço de entrega (que poderá ser modificado), o tempo de preparo (levando em consideração o item mais demorado) e o tempo médio de entrega, e solicitar a confirmação. Caso o cliente confirme o pedido, o sistema o marcará como concluído e dará baixa no estoque em todos os itens necessários a execução do pedido, incluindo os ingredientes necessários à produção de cada pizza.
- O botão **Sabores Mais Pedidos** deverá apresentar os sabores de pizzas mais solicitadas na PizzaNet, como sugestão ao cliente.
- O botão **Pedidos Anteriores** deverá apresentar uma lista de todos os pedidos já solicitados pelo cliente, permitindo que este solicite novamente um pedido já realizado, podendo realizar modificações no novo pedido, se assim o desejar.
- O botão **Opinar** só poderá ser utilizado caso o cliente tenha se autenticado. Essa opção deverá permitir que o cliente dê a sua opinião sobre o atendimento da pizzeria, referindo-se tanto à qualidade da pizza como da entrega. Dessa forma, é necessário manter informações referentes a qual funcionário fez a pizza e qual a entregou.
- Durante o registro do pedido, o sistema deverá salvar também todos os seus itens, ou seja, as pizzas e bebidas solicitadas.
- Um cliente poderá realizar muitos pedidos, no entanto, um pedido será exclusivo para um único cliente.
- Cada pedido deverá armazenar, entre outras informações, a data e a hora em que o pedido foi feito e a hora provável de sua entrega, calculada de acordo com o tempo de preparo da pizza mais demorada e o tempo médio de entrega na cidade.
- Um pedido deverá ser composto de no mínimo um item, podendo conter muitos itens. Cada item é relativo a uma pizza ou bebida, independentemente de quantidade.

- Existe uma grande quantidade de sabores, e cada sabor tem um valor específico. No entanto, caso o cliente opte por pedir uma pizza com mais de um sabor, o valor desta será calculado pelo sabor mais caro multiplicado pelo número de pedaços da pizza.
- Cada pizza consome diversas quantidades de diversos itens de estoque. Sempre que uma determinada pizza for produzida, essas quantidades devem ser diminuídas de seus respectivos itens no estoque.
- Quando qualquer pedido for entregue deverão ser conferidos os produtos e as quantidades solicitadas.
- Cada pedido é produzido por um funcionário específico, podendo ser entregue pelo mesmo ou por outro funcionário. Deverá haver uma opção na página que permita aos clientes darem sua opinião sobre a qualidade do pedido e rapidez da entrega, bem como outras sugestões. Um cliente só pode dar parecer sobre pedidos feitos por ele próprio, ou seja, o cliente terá que estar logado. Caso haja reclamações, a empresa precisa saber de quem deve cobrar.
- Sempre que um item de estoque estiver com sua quantidade abaixo ou perto da quantidade mínima, deverá ser montado um pedido para um fornecedor que venda esse tipo de produto.
- A empresa necessita de relatórios que permitam saber quais sabores de pizza são mais pedidos e em que épocas do ano ou dias da semana, para ter uma expectativa de consumo, oferecer promoções e antecipar compras de produtos de estoque.
- A empresa necessita também de relatórios que informem quais os clientes que mais consomem e em que bairros se encontram esses clientes.
- A empresa precisa saber ainda qual o consumo médio diário de cada produto, para ter uma base das quantidades a serem pedidas de cada item específico. Por exemplo, dois itens de uma mesma pizza podem consumir quantidades de itens de estoque diferentes.

A partir dessa lista de requisitos, elaboramos dois formulários, à guisa de protótipo, de caráter meramente ilustrativo, para o leitor ter uma ideia de como será a interface do sistema. As figuras 16.1 e 16.2 ilustram os formulários de escolha de pizzas e de escolha de bebidas, respectivamente.

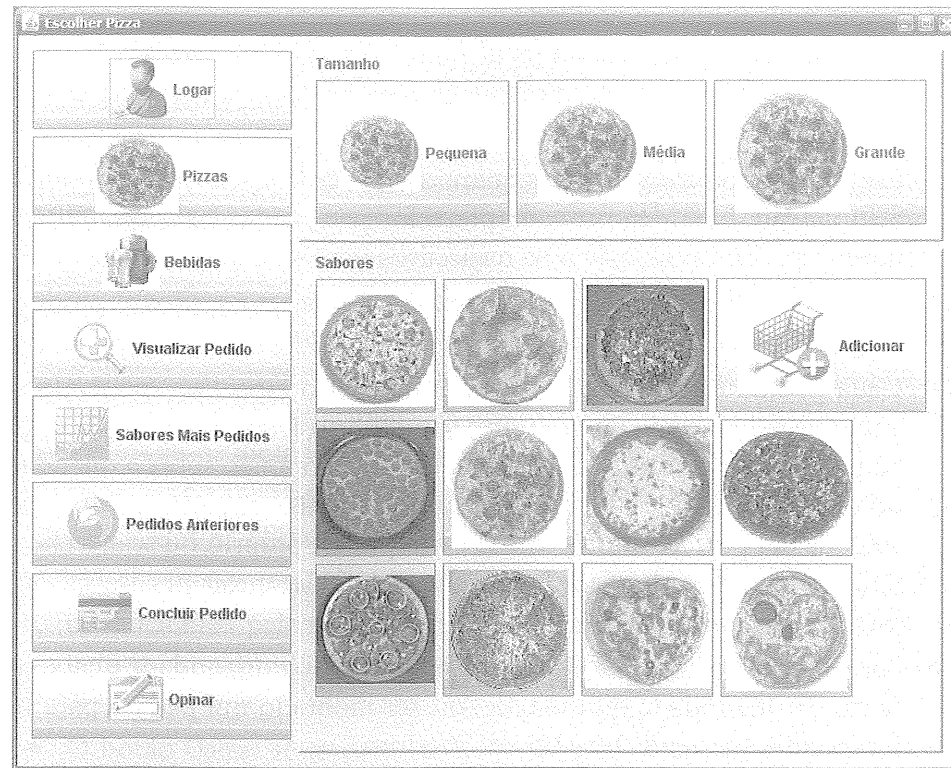


Figura 16.1 – Formulário para Escolha de Pizzas do Sistema PizzaNet.

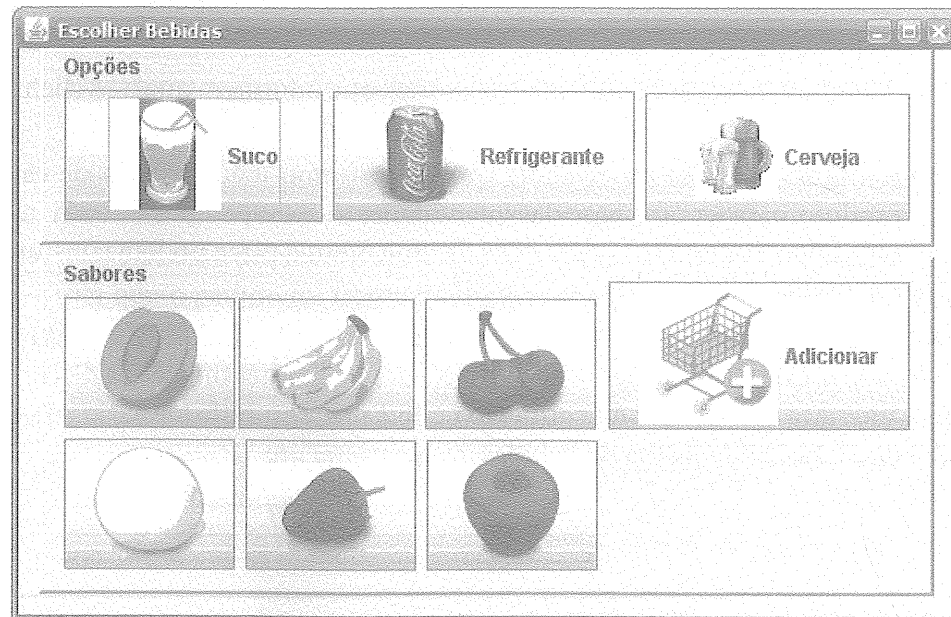


Figura 16.2 – Formulário para Escolha de Bebidas do Sistema PizzaNet.

16.2 Solução do Problema

Nas seções seguintes modelaremos uma solução para o problema apresentado na seção anterior, por meio dos diversos diagramas da UML já ensinados, sempre que estes se mostrarem úteis à modelagem desse sistema. Os diagramas serão explicados de maneira a esclarecer o que cada um deles representa.

16.2.1 Diagramas de Casos de Uso

Nesse estudo de caso decidimos modelar o sistema por meio de dois diagramas de caso de uso independentes devido ao fato de existirem dois grupos de funcionalidades diferentes, representados pelos serviços que podem ser utilizados externamente pelos clientes via Internet e os serviços internos que só podem ser manipulados pelos funcionários da empresa. Dessa forma decidimos dividir o sistema PizzaNet em dois subsistemas: o subsistema de vendas e o subsistema administrativo.

Primeiro Diagrama – Subsistema de Vendas

Na figura 16.3 apresentamos o primeiro diagrama de casos de uso ilustrando os atores e funcionalidades do subsistema de vendas, que pode ser utilizado pelos clientes da PizzaNet.

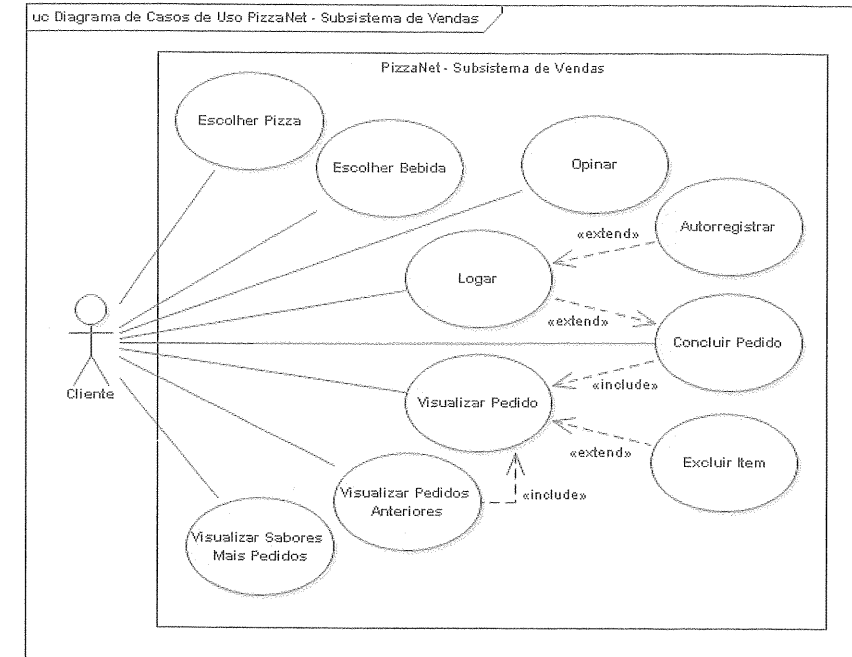


Figura 16.3 – Diagrama de Casos de Uso – Subsistema de Vendas.

O diagrama de casos de uso apresentado nessa figura contém apenas um ator, denominado **Cliente**. Esse ator representa os usuários externos, ou seja, os clientes propriamente ditos, que, por meio da internet, acessam a página da PizzaNet e realizam pedidos, contendo pizzas e/ou bebidas. O diagrama representa, ainda, as seguintes funcionalidades oferecidas ao ator **Cliente**:

- **Escolher Pizza** – este caso de uso representa o processo por meio do qual um cliente pode escolher uma pizza. Ao ter essa opção selecionada, o sistema deverá apresentar um formulário contendo duas divisões, onde a primeira apresentará os tamanhos de pizzas que a pizzeria oferece (pequeno, médio e grande) e a segunda os sabores de pizzas disponíveis. Por meio desse formulário, primeiramente o cliente deverá escolher o tamanho da pizza que deseja. A partir do tamanho escolhido, o cliente poderá escolher tantos sabores quanto os permitidos pelo tamanho. Quando tiver concluído a escolha da pizza, bastará ao cliente selecionar o botão **Adicionar** para adicionar a pizza ao pedido. O cliente poderá repetir esse processo quantas vezes quiser.
- **Escolher Bebida** – representa o processo pelo qual um cliente escolhe uma bebida, sendo um pouco semelhante ao processo de escolha de pizza. O processo é disparado quando o cliente seleciona (clica sobre) o botão **Bebidas**, o que faz com que o sistema apresente um formulário contendo duas divisões. Na primeira divisão são apresentados os tipos de bebidas oferecidos pela PizzaNet (suco, refrigerante ou cerveja), e na segunda são mostradas as bebidas pertencentes ao tipo escolhido. Por exemplo, se o cliente escolher o tipo de bebida suco, o sistema apresentará todos os sabores de suco disponíveis. Assim, primeiramente o cliente deve escolher o tipo de bebida que deseja e, em resposta, o sistema apresentará as bebidas desse tipo disponíveis, permitindo que o cliente escolha uma delas. Depois de ter escolhido o tipo e a bebida desejada, o cliente deve selecionar o botão **Adicionar** para acrescentar a bebida a seu pedido. Esse processo poderá ser repetido quantas vezes o cliente desejar.
- **Logar** – por meio dessa opção o cliente poderá autenticar-se na PizzaNet, informando seu **nome-login**, bem como sua senha. Se os mesmos estiverem corretos, o sistema logará o cliente. Isso é necessário para que o cliente possa emitir opiniões sobre pedidos anteriores, concluir um pedido ou visualizar pedidos já feitos anteriormente por ele. No entanto, caso o cliente não esteja registrado no sistema, será preciso primeiro executar o caso de uso **Autorregistrar**, que será explicado a seguir.

- **Autorregistrar** – está associado ao caso de uso **Logar** por meio de uma associação de extensão, o que significa que esse caso de uso poderá ser chamado a partir do caso de uso **Logar** quando uma condição for satisfeita. Aqui a condição é que o cliente não esteja registrado. Quando o cliente escolher se registrar, o sistema apresentará um formulário solicitando os dados do cliente, como seu nome e endereço. Depois de informar seus dados e confirmar, o cliente será cadastrado na PizzaNet.
- **Opinar** – esse serviço permite que o cliente emita opiniões sobre os pedidos feitos anteriormente por ele. Quando essa alternativa for selecionada, o sistema apresenta um formulário onde o cliente escreverá sua opinião e, se esta for confirmada, o sistema a registrará. Esse caso de uso só pode ser utilizado depois que o cliente autenticar-se no sistema.
- **Visualizar Pedido** – por meio dessa opção o cliente pode visualizar os itens escolhidos (pizzas e bebidas) para seu pedido. Essa funcionalidade apresenta todas as pizzas e bebidas escolhidas pelo cliente. Eventualmente um cliente pode querer excluir algum dos itens selecionados. Essa funcionalidade está descrita no caso de uso **Excluir Item**.
- **Excluir Item** – representa a situação em que o cliente, a partir do caso de uso **Visualizar Pedido**, deseja excluir um item de seu pedido. Para isso, ele deverá selecionar o item que deseja eliminar e escolher a opção **Excluir Item**. Quando essa opção for escolhida, o item selecionado será eliminado do pedido do cliente. Observe que existe uma associação de extensão entre o caso de uso **Excluir Item** e o **Visualizar Pedido**, o que demonstra que o primeiro caso de uso será chamado pelo segundo somente quando uma condição for satisfeita, ou seja, quando o cliente selecionar um item e pressionar o botão **Excluir Item**.
- **Visualizar Pedidos Anteriores** – por meio desse caso de uso é possível ao cliente visualizar todos os pedidos já feitos por ele. Observe que esse caso de uso tem uma associação de inclusão com o caso de uso **Visualizar Pedido**, uma vez que esse processo seleciona todos os pedidos já feitos pelo cliente, mas para apresentar cada um deles chama a rotina já descrita no caso de uso **Visualizar Pedido**.
- **Visualizar Sabores Mais Pedidos** – esse processo apresenta todos os sabores da pizzeria em ordem de sua maior predileção, ou seja, os sabores que são mais solicitados pelos clientes são apresentados primeiro.

- **Concluir Pedido** – representa o último passo para solicitar um pedido. O processo apresenta uma associação de extensão com o caso de uso Logar, o que significa que, caso o cliente não tenha se autenticado ainda, o sistema deverá executar esse caso de uso para que o cliente se logue. Observe também que o caso de uso **Concluir Pedido** tem uma relação de inclusão com o caso de uso **Visualizar Pedido**, já que é obrigatório que o cliente visualize seu pedido ainda uma vez, antes de concluí-lo. Somente depois de o cliente estar logado e ter visualizado o pedido, o sistema definirá o pedido como concluído. Quando isso ocorre, é dada baixa no estoque de todos os produtos necessários à produção do pedido.

Segundo Diagrama – Subsistema Administrativo

A seguir descreveremos o segundo diagrama de casos de uso, referente ao subsistema administrativo, apresentado na figura 16.4, onde são modeladas as funcionalidades utilizadas internamente pelos funcionários da empresa.

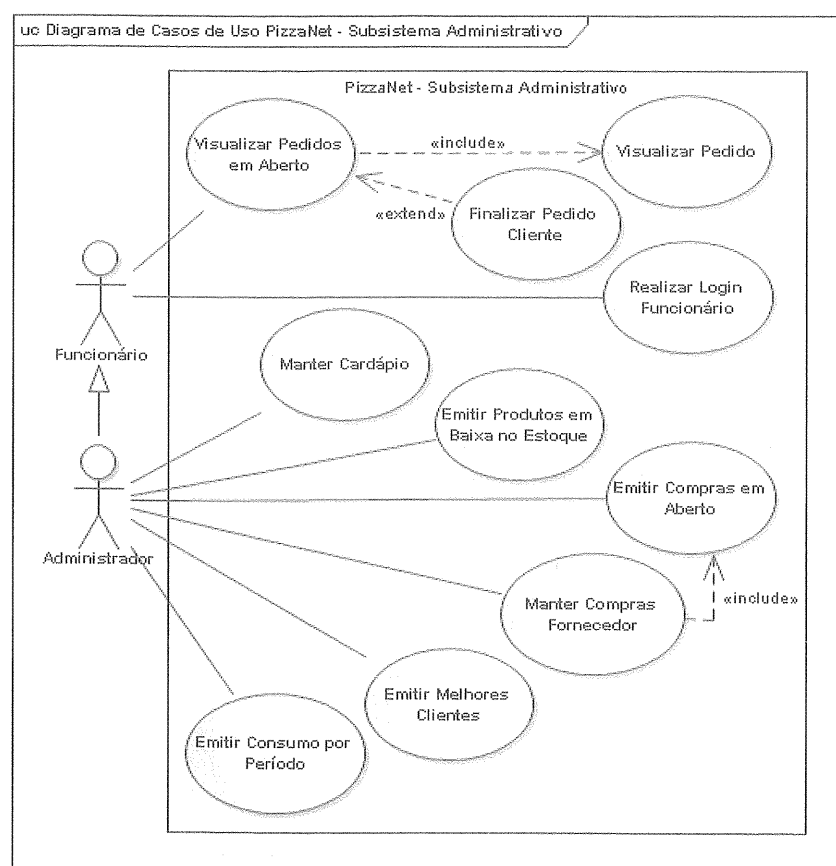


Figura 16.4 – Diagrama de Casos de Uso – Subsistema Administrativo.

Esse diagrama contém dois atores. O primeiro representa os funcionários da empresa responsáveis por atender aos pedidos dos clientes. Esse ator tem permissão para executar somente os casos de uso **Realizar Login Funcionário**, **Visualizar Pedidos em Aberto** e **Finalizar Pedido Cliente**.

Já o segundo é um ator derivado a partir do primeiro, uma vez que se trata também de um funcionário, porém com um nível de permissão maior, chamado **Administrador**. Esse ator pode executar todos os casos de uso aqui representados, incluindo os já descritos. Os outros casos de uso, que serão explicados nesta seção, têm um caráter de cunho mais administrativo e, por esse motivo, só podem ser utilizados pelo ator **Administrador**. A seguir descreveremos os casos de uso desse diagrama.

- **Realizar Login Funcionário** – representa o processo para que um funcionário se autentique no sistema. Devido ao fato de esse processo ser extremamente semelhante ao caso de uso Logar do subsistema de vendas, este será o único caso de uso que não será detalhado por meio de outros diagramas.
- **Visualizar Pedidos em Aberto** – este processo permite que um funcionário obtenha uma listagem de todos os pedidos ainda não atendidos. Observe que existe uma associação de inclusão entre esse caso de uso e o caso de uso **Visualizar Pedido**, descrito na sessão anterior. Assim, ao executar esse processo, para cada pedido em aberto é chamado o processo **Visualizar Pedido** para apresentar seus detalhes. Observe que existe ainda uma relação de extensão entre esse caso de uso e o caso de uso **Finalizar Pedido Cliente**, que representa a situação em que o pedido foi terminado pelo funcionário e este deseja defini-lo como concluído. Como esta é uma situação que só ocorre mediante a condição descrita acima, é representada por uma extensão.
- **Finalizar Pedido Cliente** – define um pedido como finalizado, determinando o funcionário que o preparou e o que o entregou. Apresenta uma associação de extensão com o caso de uso **Visualizar Pedidos em Aberto**, devido ao fato de ser chamado a partir deste, mediante a condição de que o funcionário tenha finalizado um dos pedidos listados. Quando isso ocorre, o funcionário deve selecionar o pedido na listagem e selecionar a opção finalizar pedido, o que disparará esse processo.
- **Manter Cardápio** – este é um caso de uso secundário que só pode ser executado pelo **Administrador**. Esse processo permite ao **Administrador** consultar, incluir, alterar ou excluir pizzas do cardápio da PizzaNet.

- **Emitir Produtos em Baixa no Estoque** – esse processo, que também só pode ser executado pelo Administrador, gera um relatório apresentando todos os produtos em baixa no estoque.
- **Emitir Compras em Aberto** – apresenta uma listagem ao Administrador contendo todas as compras solicitadas a fornecedores que ainda não foram entregues.
- **Manter Compras Fornecedor** – permite ao Administrador efetuar a manutenção das compras da empresa. Por meio desse processo, é possível consultar, incluir, alterar ou excluir compras. Observe que existe uma associação de inclusão entre esse caso de uso e o caso de uso anterior, determinando que, no momento em que esse caso de uso for executado, o caso de uso **Emitir Compras em Aberto** será também executado e, a partir da listagem por ele fornecida, o Administrador poderá definir uma compra como concluída, ou incluir uma nova compra.
- **Emitir Melhores Clientes** – representa o processo de emissão de um relatório de clientes, ordenado pelos clientes que mais consomem na PizzaNet.
- **Emitir Consumo por Período** – representa o relatório que informa o consumo dos itens do estoque em um determinado período fornecido pelo Administrador.

16.2.2 Documentação dos Diagramas de Casos de Uso da PizzaNet

Nesta seção o leitor poderá examinar as documentações de cada caso de uso descrito na seção anterior.

Tabela 16.1 – Documentação do Caso de Uso Escolher Pizza

Nome do Caso de Uso	Escolher Pizza
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para escolher uma pizza
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Escolher Pizza (O simples acesso à página já caracteriza esta escolha, pois esta é a opção default, mas o cliente pode selecioná-la também durante sua interação com o sistema, caso anteriormente tenha selecionado outra opção)	
	2. Apresentar tamanhos e sabores disponíveis

3. Selecionar tamanho da pizza	
	4. Permitir a escolha de sabores de acordo com o tamanho selecionado
5. Selecionar tantos sabores quantos desejados até o limite do tamanho escolhido	
6. Adicionar pizza ao carrinho de pizzas	
	7. Armazenar pizza escolhida ao pedido
Restrições/Validações	

Tabela 16.2 – Documentação do Caso de Uso Escolher Bebida

Nome do Caso de Uso	Escolher Bebida
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para adicionar bebida ao seu pedido
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Escolher Bebida	
	2. Apresentar tela para escolha de bebida
3. Escolher tipo de bebida (suco, refrigerante ou cerveja)	
	4. Apresentar todas as bebidas disponíveis para o tipo selecionado
5. Selecionar a bebida, informando a quantidade desejada, e adicioná-la ao pedido	
	6. Acrescentar a bebida escolhida ao pedido
Restrições/Validações	

Tabela 16.3 – Documentação do Caso de Uso Logar

Nome do Caso de Uso	Logar
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para logar-se ao sistema
Pré-Condições	É preciso estar cadastrado

Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Logar	
	2. Apresentar o formulário de login
3. Informar login e senha	
	4. Autenticar cliente
Fluxo Alternativo – Manutenção do Cadastro de Cliente	
Ações do Ator	Ações do Sistema
	1. Caso o cliente não esteja cadastrado, executar o módulo de Autorregistrar
Restrições/Validações	

Tabela 16.4 – Documentação do Caso de Uso Autorregistrar

Nome do Caso de Uso	Autorregistrar
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso apresenta os passos para que o cliente se cadastre
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Autorregistrar	
	2. Apresentar módulo de cadastro
3. Fornecer dados e confirmar	
	4. Registrar cliente
Restrições/Validações	

Tabela 16.5 – Documentação do Caso de Uso Opinar

Nome do Caso de Uso	Opinar
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para expressar uma opinião sobre o atendimento e qualidade da Pizzanet
Pré-Condições	É preciso estar logado

Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Opinar	
	2. Apresentar a tela para submeter uma opinião
2. Informar opinião	
	4. Registrar opinião
Restrições/Validações	

Tabela 16.6 – Documentação do Caso de Uso Visualizar Pedido

Nome do Caso de Uso	Visualizar Pedido
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para visualizar os itens de seu pedido
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Visualizar Pedido	
	2. Apresentar todas as pizzas, com seus respectivos sabores, e as bebidas escolhidas pelo cliente.
Fluxo Alternativo – Excluir Item	
Ações do Ator	Ações do Sistema
1. Selecionar item e solicitar sua exclusão	
	2. Executar o caso de uso Excluir Item.
Restrições/Validações	

Tabela 16.7 – Documentação do Caso de Uso Excluir Item

Nome do Caso de Uso	Excluir Item
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas para excluir um item do pedido
Pré-Condições	
Pós-Condições	

Fluxo Alternativo I – Excluir Pizza	
Ações do Ator	Ações do Sistema
	1. Excluir pizza
	2. Excluir itens de sabor da pizza
Fluxo Alternativo II – Excluir Bebida	
Ações do Ator	Ações do Sistema
	1. Excluir bebida
Restrições/Validações	

Tabela 16.8 – Documentação do Caso de Uso Visualizar Pedidos Anteriores

Nome do Caso de Uso	Visualizar Pedidos Anteriores
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso apresenta todos os pedidos já realizados pelo cliente.
Pré-Condições	O cliente precisa estar logado
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar a opção Visualizar Pedidos Anteriores.	
	2. Para cada pedido executar o módulo Visualizar Pedido.
	3. Apresentar pedidos.
Fluxo Alternativo – Pedir um Pedido Anterior	
Ações do Ator	Ações do Sistema
1. Opcionalmente, o cliente poderá selecionar um dos pedidos já realizados para pedi-lo novamente.	
	2. Registrar o novo pedido.
Restrições/Validações	

Tabela 16.9 – Documentação do Caso de Uso Visualizar Sabores Mais Pedidos

Nome do Caso de Uso	Visualizar Sabores Mais Pedidos
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso apresenta os sabores de pizzas mais pedidos pelos clientes.
Pré-Condições	
Pós-Condições	

Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Sabores Mais Pedidos	
	2. Selecionar os sabores solicitados nos pedidos e somá-los
	3. Apresentar sabores por ordem dos mais solicitados
Restrições/Validações	

Tabela 16.10 – Documentação do Caso de Uso Concluir Pedido

Nome do Caso de Uso	Concluir Pedido
Caso de Uso Geral	
Ator Principal	Cliente
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para concluir seu pedido
Pré-Condições	1. O cliente necessita estar logado 2. É necessário ter adicionado ao menos um item (pizza ou bebida) ao pedido
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Fechar Pedido	
	2. Executar o caso de uso Visualizar Pedido
	3. Apresentar o endereço de entrega e permitir que este seja alterado
4. Confirmar o pedido	
	5. Dar baixa nos itens do estoque necessários ao atendimento do pedido
	6. Concluir o pedido
Fluxo Alternativo I – Logar Cliente	
Ações do Ator	Ações do Sistema
	1. Caso o cliente não esteja logado, executar o módulo de Logar
Fluxo Alternativo II – Atualizar Endereço	
Ações do Ator	Ações do Sistema
1. Caso o cliente deseje, fornecer novo endereço para entrega e confirmar.	
	2. Atualizar o endereço para entrega do cliente.
Restrições/Validações	Caso o cliente exclua todos os itens do pedido, este deve ser cancelado

Tabela 16.11 – Documentação do Caso de Uso Realizar Login Funcionário

Nome do Caso de Uso	Realizar Login Funcionário
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário para logar-se ao sistema
Pré-Condições	É preciso estar registrado
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Realizar Login Funcionário	
	2. Apresentar o formulário para logar funcionários
3. Informar login e senha	
	4. Autenticar funcionário
Restrições/Validações	

Tabela 16.12 – Documentação do Caso de Uso Visualizar Pedidos em Aberto

Nome do Caso de Uso	Visualizar Pedidos em Aberto
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	
Resumo	Este caso de uso apresenta os passos para que um funcionário verifique quais pedidos se encontram em aberto
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Visualizar Pedidos em Aberto	
	2. Para cada pedido em aberto executar o caso de uso Visualizar Pedido
Restrições/Validações	
Fluxo Alternativo I – Finalizar Pedido	
Ações do Ator	Ações do Sistema
1. Escolher um pedido e selecionar a opção Finalizar Pedido	
	2. Executar o caso de uso Finalizar Pedido Cliente
Restrições/Validações	

Tabela 16.13 – Documentação do Caso de Uso Finalizar Pedido Cliente

Nome do Caso de Uso	Finalizar Pedido Cliente
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	
Resumo	Este caso de uso apresenta os passos para que um funcionário finalize um pedido em aberto
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
	1. Apresentar os funcionários disponíveis
2. Informar funcionários que prepararam e entregaram o pedido	
	3. Finalizar pedido
Restrições/Validações	É necessário haver funcionários cadastrados

Tabela 16.14 – Documentação do Caso de Uso Manter Cardápio

Nome do Caso de Uso	Manter Cardápio
Caso de Uso Geral	
Ator Principal	Administrador
Atores Secundários	
Resumo	Este caso de uso detalha os passos para que o administrador possa inserir, alterar ou excluir sabores oferecidos no cardápio da pizzaria
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Manter Cardápio	
	2. Carregar todos os sabores oferecidos pela pizzaria
	3. Carregar todos os ingredientes utilizados pela pizzaria
Restrições/Validações	É necessário haver ingredientes
Fluxo Alternativo I – Incluir Sabor	
Ações do Ator	Ações do Sistema
1. Selecionar a opção inclusão de sabor	
2. Informar sabor e ingredientes	
	3. Registrar sabor

Restrições/Validações	
Fluxo Alternativo II – Consultar Sabor	
Ações do Ator	Ações do Sistema
1. Selecionar e escolher a opção de consulta de sabor	
	2. Apresentar ingredientes do sabor
Restrições/Validações	Deve-se selecionar um sabor
Fluxo Alternativo III – Alterar Sabor	
Ações do Ator	Ações do Sistema
1. Informar alterações no sabor e/ou ingredientes	
	2. Registrar alterações
Restrições/Validações	
Fluxo Alternativo IV – Excluir Sabor	
Ações do Ator	Ações do Sistema
1. Selecionar a exclusão do sabor	
	2. Excluir sabor
Restrições/Validações	

Tabela 16.15 – Documentação do Caso de Uso Emitir Produtos em Baixa no Estoque

Nome do Caso de Uso	Emitir Produtos em Baixa no Estoque
Caso de Uso Geral	
Ator Principal	Administrador
Atores Secundários	
Resumo	Este caso de uso detalha os passos para que o administrador visualize os produtos cujas quantidades estão baixas no estoque
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Emitir Produtos em Baixa no Estoque	
	2. Apresentar ingredientes cujas quantidades estejam em baixa
	3. Apresentar bebidas cujas quantidades estejam em baixa
Restrições/Validações	

Tabela 16.16 – Documentação do Caso de Uso Emitir Compras em Aberto

Nome do Caso de Uso	Emitir Compras em Aberto
Caso de Uso Geral	
Ator Principal	Administrador
Atores Secundários	
Resumo	Este caso de uso detalha os passos para que o funcionário obtenha uma listagem das compras ainda não atendidas pelo fornecedor
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Emitir Compras em Aberto	
	2. Selecionar compras cuja situação esteja em aberto
	3. Apresentar compras em aberto detalhando os itens solicitados
Restrições/Validações	

Tabela 16.17 – Documentação do Caso de Uso Manter Compras Fornecedor

Nome do Caso de Uso	Manter Compras Fornecedor
Caso de Uso Geral	
Ator Principal	Administrador
Atores Secundários	
Resumo	Este caso de uso detalha os passos para solicitar pedidos de compras a fornecedores
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Manter Compras Fornecedor	
	2. Executar Caso de Uso Emitir Compras em Aberto
	3. Carregar os ingredientes utilizados
	4. Carregar as bebidas comercializadas
	5. Apresentar tela de manutenção de compras permitindo ao usuário finalizar uma compra em aberto ou gerar uma nova compra
Fluxo Alternativo I – Finalizar Compra	
Ações do Ator	Ações do Sistema
1. Selecionar uma compra em aberto e solicitar seu fechamento	

	2. Finalizar compra
Restrições/Validações	É preciso selecionar uma compra
Fluxo Alternativo II – Gerar Nova Compra	
Ações do Ator	Ações do Sistema
1. Escolher a opção para gerar uma nova compra	
2. Selecionar ingredientes e quantidades a comprar	
3. Selecionar bebidas e quantidades a comprar	
4. Selecionar fornecedor	
5. Confirmar compra	
	6. Registrar compra
Restrições/Validações	Deve-se selecionar ao menos um ingrediente ou bebida

Tabela 16.18 – Documentação do Caso de Uso Emitir Melhores Clientes

Nome do Caso de Uso	Emitir Melhores Clientes
Caso de Uso Geral	
Ator Principal	Administrador
Atores Secundários	
Resumo	Este caso de uso detalha os passos para apresentar os melhores clientes, considerando-se o maior valor gasto pelos clientes.
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Emitir Melhores Clientes	
	2. Selecionar clientes
	3. Selecionar os pedidos de cada cliente
	4. Totalizar os valores das pizzas dos pedidos
	5. Totalizar os valores das bebidas dos pedidos
	6. Apresentar os clientes por ordem de maior valor gasto, junto com o bairro em que estes residem
Restrições/Validações	

Tabela 16.19 – Documentação do Caso de Uso Emitir Consumo por Período

Nome do Caso de Uso	Emitir Consumo por Período
Caso de Uso Geral	
Ator Principal	Administrador
Atores Secundários	
Resumo	Este caso de uso apresenta os passos para que o administrador verifique o consumo dos ingredientes em um determinado período
Pré-Condições	
Pós-Condições	
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Emitir Consumo por Período	
	2. Solicitar períodos desejados
3. Informar períodos	
	4. Selecionar todas as pizzas dos pedidos realizados no período
	5. Totalizar os ingredientes de cada sabor pedido
	6. Apresentar consumo no período solicitado
Restrições/Validações	

16.2.3 Diagrama de Classes – Modelo de Domínio

Nesta seção apresentaremos o modelo de domínio da solução para o sistema de pizzaria online (Figura 16.5).

Descreveremos as classes que compõem esse diagrama juntamente com suas associações, métodos e atributos. Embora já tenhamos afirmado isso no capítulo sobre o diagrama de classes, acreditamos ser útil salientar novamente que, a rigor, deve haver um método get e um método set para cada atributo de uma classe. No entanto, essa modelagem enfoca o sistema em um nível mais alto, assim consideramos impraticável e mesmo desnecessário inserir um conjunto de métodos óbvios em classes que contenham um grande número de atributos. Dessa forma definimos métodos como “registrar” ou “consultar” para inserir ou retornar conjuntos de atributos sempre que o processo não exigir métodos exclusivos para um atributo específico. Por esse motivo, muitos métodos aqui descritos retornam um atributo do tipo String, que poderá conter os valores de um conjunto de atributos necessários ao processo.

consultado. Obviamente, é necessário existir um método para registrar novos bairros, no entanto, uma vez que o diagrama tem um conjunto muito grande de informações, procuramos definir somente os métodos realmente importantes. Muitos clientes podem morar em um determinado bairro, como demonstra a associação *mora*, mas um cliente específico só pode morar em um único bairro.

6. Cargo

Essa classe armazena os cargos assumidos pelos funcionários da empresa. Seu único atributo é a descrição do cargo do tipo *String*.

7. Funcionario

A classe *Funcionario* armazena as informações dos funcionários que trabalham ou trabalharam na empresa. Seus atributos contêm o CPF do funcionário, do tipo *long*, o nome do tipo *String*, e o salário do tipo *double*. O único método aqui descrito é o *conFun*, que serve para consultar um funcionário, retornando uma *String* contendo seu nome. Observe que essa classe tem uma associação com a classe anterior, que determina que um funcionário deve ter um único cargo, mas um mesmo cargo pode ser atribuído a muitos funcionários.

8. Pedido

Essa classe armazena as informações relacionadas aos pedidos solicitados pelos clientes. Observe que a classe tem duas associações com a classe *Funcionario*, onde um funcionário pode preparar e/ou entregar muitos pedidos, mas um pedido só pode ser preparado por um funcionário específico e, da mesma forma, só pode ser entregue por um determinado funcionário (não necessariamente o mesmo). A classe *Pedido* tem também uma associação com a classe *Cliente*, que determina que um cliente pode solicitar muitos pedidos, mas um pedido deve estar associado somente a um cliente. Há ainda outras associações dessa classe com outras classes, que serão explicadas ao longo desta seção.

A classe *Pedido* contém os atributos número do pedido, do tipo *long*; a hora em que o pedido foi solicitado, do tipo *Time*; a data em que este foi realizado, do tipo *Date*; e a situação do pedido, do tipo *int*, que determina se o pedido ainda não foi atendido, se já foi finalizado ou se já foi entregue. É preciso destacar que os tipos *Time* e *Date* referem-se a classes que devem ser implementadas pela linguagem ou estar contidas no sistema. Essa classe tem ainda os seguintes métodos:

Método	Descrição
<i>regPed</i>	Permite registrar um novo pedido. O método não recebe nenhum parâmetro, uma vez que a hora e a data são tomadas do sistema, o valor inicial da situação do pedido é 0 e o número do pedido (um <i>long</i>) é retornado pelo método.
<i>conPed</i>	Permite a consulta de um pedido, retornando uma <i>String</i> com os seus dados.
<i>visPedAberto</i>	Retorna um <i>long</i> , contendo o número do pedido, para cada pedido cuja situação contenha o valor 0, que significa que o pedido ainda não foi atendido.
<i>selPed</i>	Seleciona um conjunto de pedidos cujas datas estejam dentro do período fornecido pelo cliente. Retorna um <i>long</i> , contendo o número de cada pedido encontrado.
<i>selPedCli</i>	Seleciona todos os pedidos de um cliente específico, retornando um <i>double</i> contendo o valor total de todos os pedidos. É utilizado no processo de emissão de melhores clientes.
<i>confirmaPed</i>	Define um pedido como confirmado, ou seja, estabelece que o cliente confirmou o pedido. Quando isso ocorre o método define a situação do pedido como 1, que, por convenção significa que o pedido foi confirmado e deve ser atendido. Esse método retorna um <i>int</i> , cujo valor pode significar que o método foi concluído com sucesso, se armazenar 1, ou que ocorreu algum erro, se armazenar 0.
<i>finalizaPed</i>	Define um pedido como finalizado, ou seja, determina que um funcionário terminou um pedido anteriormente em aberto. O método define a situação do pedido como 2, que por convenção determina que o pedido foi finalizado. Esse método retorna um <i>int</i> , que determina se o método foi concluído com sucesso (1) ou não (0).

9. Tamanho

Essa classe armazena os tamanhos disponibilizados pela *PizzaNet*. Seus atributos são a descrição do tamanho, do tipo *String*; o número de pedaços e a quantidade de sabores, ambos do tipo *int*. Essa classe tem ainda os seguintes métodos:

Método	Descrição
<i>conDesTam</i>	Retorna uma <i>String</i> contendo a descrição do tamanho (se é pequeno, médio ou grande).
<i>conNroPedacos</i>	Retorna um inteiro contendo o número de pedaços do tamanho.
<i>conQtdSab</i>	Retorna um inteiro contendo o número de sabores que uma pizza desse tamanho pode conter.

10. Pizza

Essa classe armazena as informações das pizzas solicitadas em um determinado pedido. Observe que existe uma associação de composição entre a classe Pedido e a classe Pizza, o que significa que os objetos da classe Pedido são objetos-todo, cujas informações devem ser complementadas pelas informações contidas em objetos-parte da classe Pizza (essas informações devem ser complementadas também por objetos-parte da classe Item_Bebida, que será explicada mais adiante). Dessa maneira, como podemos observar pela associação de composição entre essas duas classes, um pedido pode conter muitas pizzas, mas uma pizza deve estar contida única e exclusivamente em um pedido.

É preciso notar também que existe uma associação entre essa classe e a classe tamanho, determinando que uma pizza pode estar associada a somente um tamanho, mas um tamanho pode estar relacionado a muitas pizzas. Existe ainda outra associação que será explicada quando abordarmos a próxima classe.

A classe Pizza contém os atributos quantidade de sabores, do tipo int, e valor da pizza, do tipo double. O leitor poderá se perguntar o porquê do atributo quantidade de sabores, uma vez que essa informação pode ser puxada da classe Tamanho relacionada à classe Pizza. Isso estaria correto, mas pode acontecer de um cliente solicitar uma pizza grande com um único sabor. Por esse motivo optamos por inserir esse atributo nessa classe. O leitor poderá inquirir também por que existe o atributo valor da pizza, uma vez que este é calculado, como demonstra a barra ao lado de sua visibilidade. Optamos por inserir esse atributo para facilitar os processos que pesquisarão as pizzas dos pedidos, diminuindo o número de operações a ser realizadas e as classes a ser consultadas.

Essa classe contém ainda os seguintes métodos:

Método	Descrição
adiPizza	Serve para adicionar uma pizza ao pedido. Retorna um inteiro que determina se o método foi executado com sucesso ou não.
conPizza	Permite consultar uma pizza de um pedido e retorna uma String contendo os dados da pizza consultada.
excPizza	Por meio desse método é possível excluir uma pizza de um pedido. O método retorna um inteiro que conterá um valor verdadeiro (1) caso o método consiga excluir a pizza ou falso (0), em caso contrário.
selPizza	Seleciona todas as pizzas de um pedido, retornando uma String com os dados de cada pizza.
conValPizza	Permite consultar o valor de uma pizza específica, retornando um double contendo seu valor.

11. Item_Sabor

Essa classe armazena as informações sobre os sabores escolhidos para uma pizza. Observe que existe uma associação de composição entre a classe Pizza e a classe Item_Sabor. Isso significa que os objetos da classe Pizza são também objetos-todo em relação aos objetos da classe Item_Sabor, que desempenham o papel de objetos-parte nessa associação, uma vez que estes devem complementar as informações de um objeto da classe Pizza, fornecendo os dados de cada sabor escolhido para uma pizza específica.

Essa classe contém o atributo valor, do tipo double. Alguém poderia se perguntar se esse atributo é realmente necessário, uma vez que este pode ser puxado da classe Sabor, que explicaremos a seguir. Esse atributo é realmente necessário porque o valor de um sabor pode ser alterado, o que deturparia as informações relativas ao valor do sabor na época em que o pedido foi feito.

A classe Item_Sabor contém ainda o seguintes métodos:

Método	Descrição
adItemSabor	Permite adicionar um sabor a uma pizza. Retorna um inteiro determinando o sucesso ou não do método.
conItemSabor	Consulta um sabor específico de uma pizza, retornando uma String contendo seus dados.
excItemSabor	Permite excluir um sabor de uma pizza, retornando um inteiro que determina o sucesso ou não dessa operação.
totalizaSabor	Utilizado para totalizar quantas vezes um sabor foi pedido. O método retorna um long contendo esse total.
selItemSabor	Chamado pelo método selPizza para selecionar todos os objetos da classe Item_Sabor associados à pizza. O método retorna uma String com os dados desses objetos.

12. Sabor

Essa classe armazena as informações relacionadas aos sabores oferecidos pela pizzaria. Seus atributos são o nome do sabor, do tipo String, e o valor, do tipo double. Observe que essa classe tem uma associação com a classe Item_Sabor, que determina que um objeto da classe Sabor pode estar associado a muitos objetos da classe Item_Sabor, mas um objeto da classe Item_Sabor só pode estar relacionado a um objeto da classe Sabor. A classe contém ainda os métodos:

Método	Descrição
regSab	Permite o registro de um novo sabor. Retorna um inteiro que, se contiver verdadeiro (1), indicará que foi possível registrar o novo sabor, caso contrário, determinará que algum erro ocorreu.

Método	Descrição (cont.)
excSabor	Permite a exclusão de um sabor. Retorna um inteiro para determinar o sucesso ou não da operação.
se1Sabor	Usado durante o processo de conclusão de um pedido, para auxiliar a dar baixa nos ingredientes utilizados para a produção dos sabores. Sua função é selecionar todos os sabores solicitados no pedido. O método retorna um inteiro determinando se o método foi concluído com sucesso ou não.
conValSabor	Permite consultar o valor de um sabor, retornando um double contendo seu valor.
conDesSabor	Permite consultar a descrição do sabor, retornando uma String contendo sua descrição.
conIngSabor	Permite consultar os ingredientes necessários ao preparo de um sabor. Retorna uma String contendo a listagem dos ingredientes necessários.

O leitor talvez se pergunte por que é necessário a classe Item_Sabor e se sua função não poderia ser assumida pela classe Sabor. Acontece, porém, que uma pizza pode ter muitos sabores, e um sabor pode estar contido em muitas pizzas. Sendo assim, é necessário haver uma classe intermediária entre as classes Pizza e Sabor que armazene as informações dos sabores de uma pizza específica.

13. Ingrediente

Essa classe armazena as informações referentes aos ingredientes necessários para preparar os sabores oferecidos pela PizzaNet. Seus atributos são a descrição do ingrediente, a unidade do ingrediente, ambos do tipo String, além da quantidade em estoque e a quantidade mínima do produto (se a quantidade do ingrediente estiver igual ou abaixo do mínimo deve ser feita uma solicitação de compra a um fornecedor), ambos do tipo double.

Método	Descrição
conIng	permite consultar um ingrediente específico, retornando uma String com sua descrição.
conIngBaixa	retorna uma String contendo as informações de todos os ingredientes cuja quantidade em estoque estiver igual ou abaixo do mínimo.
baixaIng	diminui a quantidade necessária à produção de um sabor da quantidade em estoque de um ingrediente. Retorna um inteiro que determina o sucesso ou não da operação.

14. Item_Ingrediente

Esta é uma classe intermediária que contém as informações de cada ingrediente necessário à produção de um sabor. Essa classe justifica-se uma

vez que um sabor pode estar relacionado a muitos ingredientes e um ingrediente pode estar associado a muitos sabores. Assim, cada objeto dessa classe está associado a um objeto da classe Ingrediente e a um objeto da classe Sabor, mas um objeto da classe Sabor pode estar associado a muitos objetos da classe Item_Ingrediente, o mesmo ocorrendo em relação aos objetos da classe Ingrediente.

Essa classe armazena o atributo quantidade do tipo double, que estabelece a quantidade necessária de um ingrediente específico para um sabor específico. A classe tem ainda os seguintes métodos:

Método	Descrição
regItemIng	Permite registrar um novo item de ingrediente.
excItemIng	Permite excluir um item de ingrediente.
conItemIng	Permite consultar um item de ingrediente, retornando um double contendo sua quantidade.
se1ItemIng	Permite selecionar os itens de ingrediente de um determinado sabor.

15. Tipo_Bebida

Essa classe armazena os tipos de bebida oferecidos pela PizzaNet, ou seja, suco, refrigerante e cerveja. Seu único atributo é o nome, do tipo String, e seu único método, ConTipBeb, permite consultar um tipo de bebida, retornando uma String contendo seu nome.

16. Bebida

Essa classe armazena os dados de todas as bebidas comercializadas pelo sistema. Note que esta classe está associada à classe Tipo_Bebida. A multiplicidade dessa associação determina que uma bebida deve estar associada a somente um tipo de bebida, no entanto, um tipo de bebida pode estar associado a muitas bebidas. Os atributos dessa classe são a descrição da bebida, do tipo String; a quantidade em estoque e a quantidade mínima da bebida, ambos do tipo long; e o valor da bebida, do tipo double.

Essa classe contém os seguintes métodos:

Método	Descrição
conDesBeb	Permite consultar a descrição de uma bebida, retornando uma String contendo a descrição.
conValBeb	Permite consultar o valor de uma bebida, retornando um double contendo o valor em questão.
baixaBebida	Diminui (baixa) a quantidade solicitada em um pedido da quantidade em estoque de uma bebida. Retorna verdadeiro, se a operação pôde ser realizada, e falso, caso contrário.

Método	Descrição (cont.)
conBebBaixa	Consulta todas as bebidas com a quantidade em estoque abaixo da quantidade mínima, retornando uma String com os dados de cada bebida em baixa.

17. Item_Bebida

Essa é uma classe intermediária posicionada entre a classe Pedido e a classe Bebida. É necessária para armazenar as bebidas solicitadas em um pedido específico, devido ao fato de que um pedido pode incluir muitas bebidas e uma bebida pode estar presente em muitos pedidos, não havendo espaço em nenhuma das duas classes para armazenar as informações das bebidas do pedido, já que não é possível saber quantas bebidas terá um pedido, nem quantos pedidos solicitarão uma bebida. Mesmo que se reservasse um espaço grande em qualquer das duas classes, este poderia ser atingido e, enquanto isso não ocorresse um grande número de atributos, seria deixado em branco.

Observe que a classe Pedido tem uma associação de composição com a classe Item_Bebida. Isso significa que as informações de um objeto-todo da classe Pedido precisam ser complementadas pelas informações contidas nos objetos-parce da classe Item_Bebida. Além disso, essa associação nos passa a informação de que um objeto Item_Bebida está associado a um único objeto da classe Pedido e só deverá ser excluído também se o pedido a que estiver associado for destruído.

A classe Item_Bebida também está associada à classe Bebida, e a multiplicidade dessa associação passa a informação de que uma bebida pode estar associada a muitos itens de bebida, mas um item de bebida tem que estar associado a somente uma bebida.

Essa classe tem como atributo somente o valor do item de bebida do tipo double. A exemplo da classe Item_Sabor, armazenamos o valor da bebida na época em que o pedido foi feito, já que o valor armazenado na classe Bebida pode ser atualizado. Assim, se puxássemos o valor da classe bebida, o total apresentado em um pedido consultado poderia ser diferente do real, se tivesse ocorrido algum aumento no valor da bebida.

A classe Item_Bebida contém ainda os métodos:

Método	Descrição
adItemBebida	Permite adicionar uma nova bebida a um pedido, retornando verdadeiro, se o método foi executado com sucesso, ou falso, em caso contrário.

Método	Descrição (cont.)
conValBebida	Consulta o valor de uma bebida, retornando um double contendo seu valor.
conItemBebida	Consulta um item de bebida, retornando uma String com seus dados. É usado em conjunto com o método conDesBeb, já explicado na classe anterior.
excItemBebida	Permite excluir um item de bebida do pedido. O método retorna verdadeiro (1), se for finalizado com sucesso, ou falso (0), em caso contrário.

18. Fornecedor

Essa classe armazena as informações dos fornecedores dos quais a pizzaria compra ingredientes e bebidas. Seus atributos são nome, endereço, bairro, cidade, telefone e e-mail do tipo String, além do CEP do tipo long. A classe contém ainda o método conFor que permite consultar um fornecedor, retornando uma String com seus dados.

19. Compra

Essa classe contém as informações relativas aos pedidos de compras solicitados aos fornecedores. A classe contém os atributos data, do tipo Date (que se refere a uma classe), que armazena a data em que a compra foi solicitada, e situação, do tipo int, que armazena 0 (seu valor inicial), caso a compra ainda não tenha sido entregue, ou 1, caso contrário.

Essa classe tem uma associação com a classe Fornecedor, que informa que a um fornecedor podem ter sido solicitadas muitas compras, mas que uma compra está associada a somente um fornecedor. Há também uma associação com a classe Funcionario, que determina que um funcionário pode encomendar muitas compras, mas que uma compra só pode estar associada a um funcionário. Há ainda outras associações que serão explicadas quando da explanação de outras classes.

Essa classe contém os seguintes métodos:

Método	Descrição
regCompra	Permite registrar uma nova compra, retornando 1 (verdadeiro), se for possível registrar a nova compra, ou falso (0), caso contrário.
comprasAbertas	Retorna uma String contendo os dados de todas as compras ainda não entregues pelos fornecedores.
fecharCompra	Permite definir uma compra como concluída, ou seja, uma compra que foi entregue pelo fornecedor, de acordo com o que foi solicitado. Esse método muda o valor do atributo situacao para 1, significando que a compra está fechada.

20. Item_Compra_Ing

Esta é uma classe intermediária que armazena as informações dos ingredientes que foram solicitados em uma determinada compra. Observe que existe uma associação de composição entre a classe Compra e a classe Item_Compra_Ing, determinando que um objeto da classe Compra é um objeto-todo cujas informações precisam ser complementadas pelos objetos-parte da classe Item_Compra_Ing. Assim, um objeto dessa classe precisa estar associado a um único objeto da classe Compra, embora este possa estar associado a muitos objetos da classe Item_Compra_Ing.

Há ainda uma associação dessa classe com a classe Ingrediente, que determina que um objeto da classe Ingrediente pode estar associado a muitos objetos da classe Item_Compra_Ing, mas que um objeto da classe Item_Compra_Ing precisa estar vinculado, nessa associação, a somente um objeto da classe Ingrediente.

Essa classe tem como atributo unicamente a quantidade do ingrediente solicitada em cada compra, do tipo `double`. Além disso, apresenta ainda os seguintes métodos:

Método	Descrição
<code>regIngCompra</code>	Permite instanciar um novo objeto dessa classe, retorna verdadeiro, se for executado com sucesso, ou falso, caso contrário.
<code>sel_ItemCompraIng</code>	Seleciona cada ingrediente solicitado em uma determinada compra, retornando uma <code>String</code> com seus dados.

21. Item_Compra_Beb

Esta é também uma classe intermediária, posicionada entre as classes Compra e Bebida, com uma função bastante semelhante à classe Item_Compra_Ing. Essa classe armazena as informações das bebidas que foram solicitadas em uma determinada compra. O leitor notará que existe também uma associação de composição entre a classe Compra e a classe Item_Compra_Beb, o que significa que as informações dos objetos-parte dessa classe devem também complementar as informações dos objetos-todo da classe Compra. Essa associação também informa que um objeto da classe Item_Compra_Beb necessita estar vinculado, nessa associação, a somente um objeto da classe Compra, mas que um objeto da classe Compra pode estar associado a muitos objetos da classe Item_Compra_Beb.

Essa classe tem também uma associação com a classe Bebida, e essa associação informa que um objeto da classe Bebida pode estar associado a muitos objetos da classe Item_Compra_Beb, mas que um objeto da classe Item_Compra_Beb precisa estar vinculado, nessa associação, a somente um objeto da classe Bebida.

Essa classe tem como atributo unicamente a quantidade da bebida solicitada em cada compra, do tipo `double`. Além disso, contém ainda os seguintes métodos:

Método	Descrição
<code>regBebCompra</code>	Permite instanciar um novo objeto dessa classe, retorna verdadeiro, se for executado com sucesso, ou falso, caso contrário.
<code>sel_ItemCompraBeb</code>	Seleciona cada bebida solicitada em uma determinada compra, retornando uma <code>String</code> com seus dados.

16.2.4 Diagrama de Objetos

Nesta seção apresentaremos, na figura 16.6, um diagrama de objetos referente ao modelo de domínio apresentado na seção anterior. O objetivo desse diagrama é meramente ilustrativo, com o intuito de aclarar como estarão organizados os objetos do sistema quando instanciados, como aliás é o objetivo de todo diagrama de objetos.

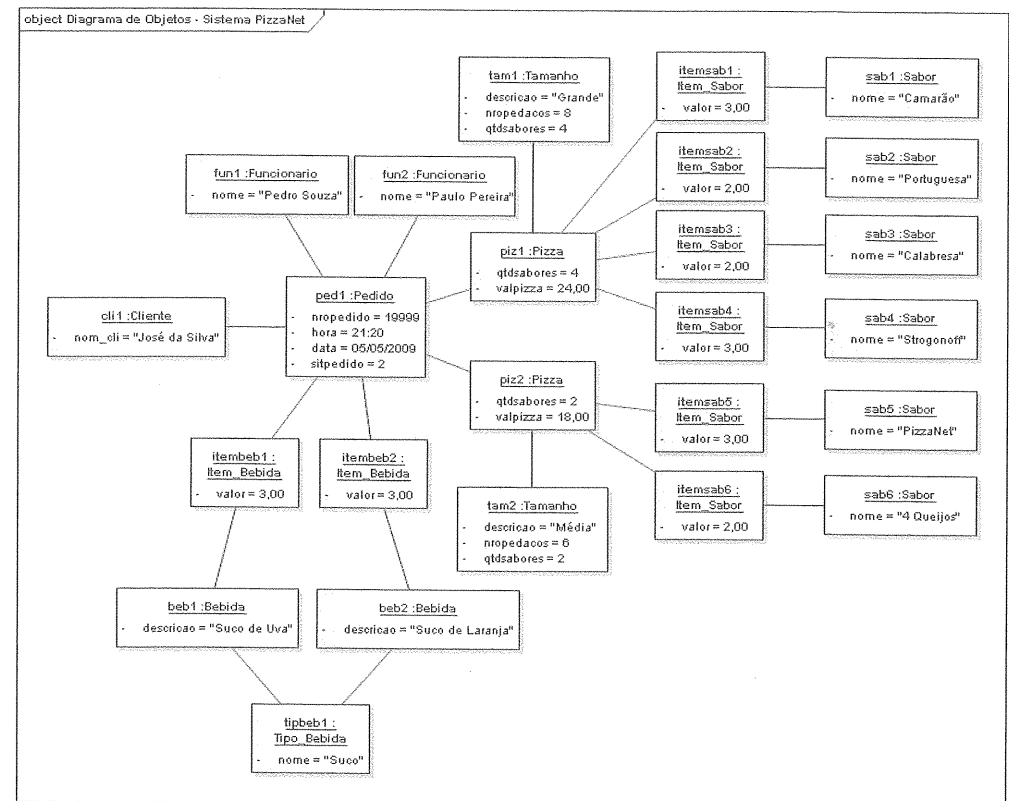


Figura 16.6 – Diagrama de Objetos PizzaNet.

Nesse exemplo ilustramos um pedido que já foi finalizado, como podemos observar pelo valor 2 do atributo `sitpedido`. O pedido em questão inclui duas pizzas e duas bebidas. A primeira pizza tem quatro sabores e a segunda dois. As duas bebidas são do tipo suco, mas cada uma se refere a um suco diferente. Finalmente, o pedido foi atendido por dois funcionários, sendo que um preparou e outro o entregou.

O leitor notará que muitos objetos não possuem todos os atributos contidos nas classes a partir da qual foram instanciados, optamos por essa abordagem para não deixar o diagrama extenso demais, definindo o valor somente dos atributos mais importantes.

Talvez o leitor se pergunte por que o valor do objeto `piz1` é 24,00 quando a soma dos valores dos objetos `Item_Sabor` a ele associados não chega a esse valor. Isso ocorre porque o valor da pizza é calculado pela multiplicação do valor do sabor mais caro pelo número de pedaços da pizza (que neste caso são respectivamente 3,00 e 8).

16.2.5 Diagrama de Pacotes da PizzaNet

Nesta seção apresentaremos, na figura 16.7, o diagrama de pacotes do sistema PizzaNet, onde podemos perceber que este é composto por dois subsistemas, o subsistema de vendas e o subsistema administrativo. Podemos perceber também que o subsistema administrativo tem uma dependência com relação ao subsistema de vendas, uma vez que é necessário haver pedidos para que os módulos administrativos possuam dados que justifiquem sua execução.

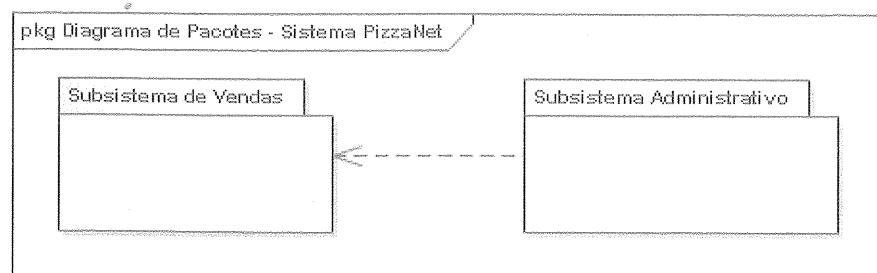


Figura 16.7 – Diagrama de Pacotes PizzaNet.

16.2.6 Diagramas de Sequência da PizzaNet

Nesta seção apresentaremos os diagramas de sequência referentes a esse estudo de caso, que também são uma forma de documentar cada caso de uso apresentado no início deste capítulo. Talvez alguns dos métodos apresentados nos diagramas a seguir poderiam ser simplificados, e algumas vezes poderia não

ser necessária a chamada de outros métodos para complementá-los, podendo tais métodos ser absorvidos em alguns casos pelos métodos que os chamaram. No entanto, optamos por uma abordagem mais detalhada para facilitar a compreensão dos diagramas, mesmo porque os diagramas não estão baseados em nenhuma linguagem de programação específica, o que, aliás, é o correto.

Além disso, como já foi dito anteriormente, a rigor deve haver um método `get` e um método `set` para cada atributo de uma classe, mas isso é impraticável em processos que contenham um número grande de atributos, além de desnecessário, uma vez que os diagramas são modelos de nível mais alto. Assim utilizamos algumas vezes métodos como “registrar” para instanciar um objeto ou “consultar” para retornar todas as informações de uma instância.

Diagrama de Sequência Escolher Pizza

A figura 16.8 apresenta o detalhamento do processo Escolher Pizza por meio de um diagrama de sequência.

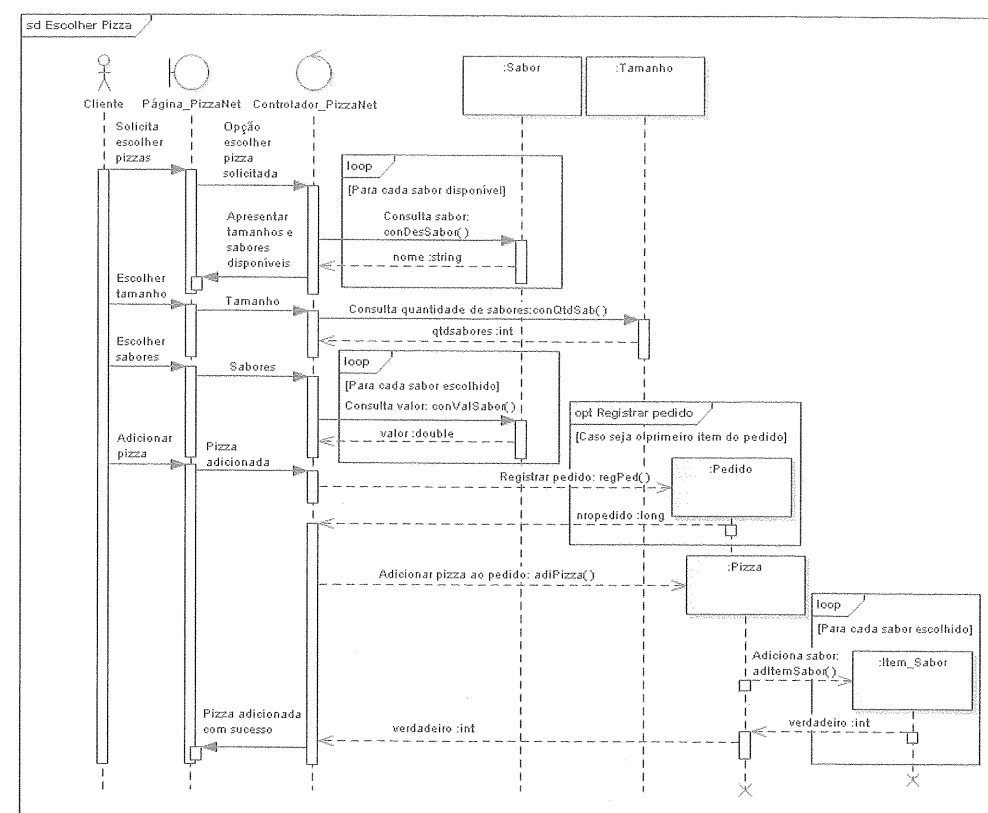


Figura 16.8 – Diagrama de Sequência Escolher Pizza.

Esse processo inicia-se quando o cliente acessa a página da PizzaNet, por esta ser a opção default; ou quando ele clica sobre o botão Pizzas da interface. Ao ser avisado da ocorrência desse evento, o controlador dispara o método `conDesSabor` em cada objeto da classe `Sabor`, para retornar sua descrição. Observe que existe um fragmento combinado do tipo `loop`, que demonstra que isso ocorre por meio de um laço. A partir das descrições retornadas pelas chamadas desse método, o controlador manda carregar na página os tamanhos e os sabores disponíveis.

O cliente deve então selecionar o tamanho desejado da pizza, que é repassado para o controlador que dispara o método `conQtdSab` para retornar a quantidade de sabores que podem ser pedidos para o tamanho escolhido. Depois disso o cliente poderá escolher os sabores desejados para a pizza, simplesmente clicando sobre o sabor. A escolha desses sabores é repassada para o controlador que irá executar um laço, disparando o método `conValSabor` para cada sabor escolhido, recuperando o valor do sabor. Quando o cliente tiver terminado de escolher os sabores da pizza, bastará pressionar o botão adicionar. Esse evento obrigará o controlador a realizar um teste para determinar se a pizza escolhida é o primeiro item do pedido, conforme demonstra o fragmento combinado do tipo `opt`. Caso o teste seja positivo, o controlador disparará o método `regPed` que instanciará um novo objeto da classe `Pedido` e retornará o número do mesmo.

O controlador então executará o método `adiPizza` para adicionar a nova pizza ao pedido. Este é um método criador, que instanciará um novo objeto da classe `Pizza` e chamará o método `adiItemSabor` para instanciar os objetos da classe `Item_Sabor` associados à pizza. O método será disparado tantas vezes quantos forem os sabores escolhidos, como demonstra o fragmento combinado do tipo `loop`. Esse método é também um método criador. Se o retorno desses métodos for verdadeiro, será apresentada a mensagem de “Pizza adicionada com sucesso”.

Diagrama de Sequência Escolher Bebida

Esse processo se inicia quando o cliente pressiona o botão **Bebidas** na página da PizzaNet. Esse evento é repassado pela página ao controlador que ordena à página que seja apresentado o formulário para escolha de bebidas, onde serão apresentados os três tipos de bebidas oferecidos pela pizzeria: suco, refrigerante e cerveja.

Nesse formulário o cliente deve escolher o tipo de bebida que deseja. Esse evento faz com que o controlador consulte o tipo de bebida escolhido, por meio da chamada ao método `conTipBeb` e, após isso, o controlador iniciará um laço, representado pelo fragmento combinado do tipo `loop`, onde executará o método

`conDesBeb` e logo após o método `conValBeb` para retornar a descrição e o valor de cada bebida do tipo escolhido. Em poder dessas informações, o controlador mandará atualizar a página, apresentando todas as bebidas do tipo escolhido ao cliente.

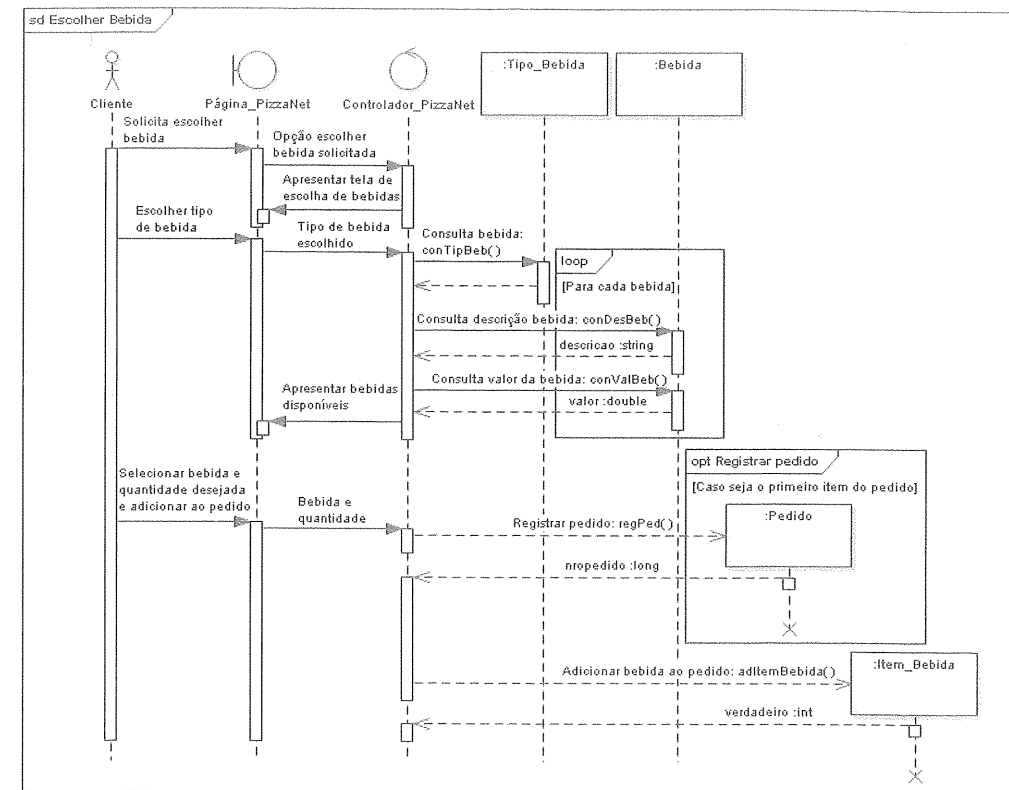


Figura 16.9 – Diagrama de Sequência Escolher Bebida.

Em seguida, o cliente deverá selecionar a bebida e a quantidade desejada e pressionar o botão **Adicionar**, para acrescentar a bebida ao pedido. Esse evento fará com que o controlador novamente realize o teste já descrito no processo anterior, para determinar se este é o primeiro item do pedido do cliente, conforme demonstra o fragmento combinado do tipo `loop`. Caso isso seja verdadeiro, o controlador irá disparar o método `regPed` em um objeto da classe `Pedido`, instanciando-o e retornando o número do novo pedido.

Após esse teste, o controlador chamará o método `adItemBebida` para instanciar um novo objeto da classe `Item_Bebida`, relativo à bebida que foi adicionada ao pedido. O retorno verdadeiro (na verdade o valor 1) significará que a bebida foi adicionada com sucesso.

Diagrama de Sequência Logar

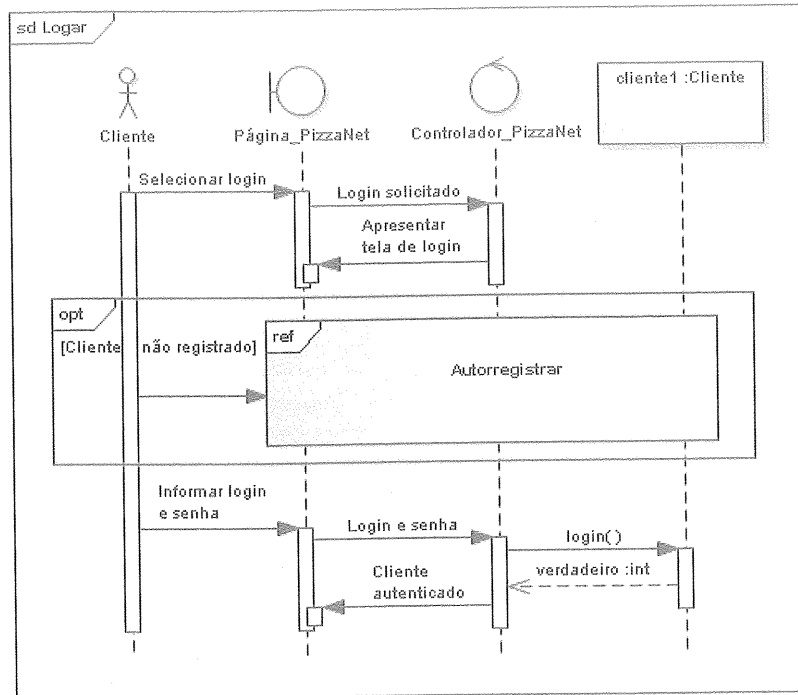


Figura 16.10 – Diagrama de Sequência Logar.

Esse processo pode ser chamado pelo cliente quando este clicar sobre o botão **Logar** ou pode ser chamado pelo processo **Concluir Pedido**, caso o cliente não tenha se autenticado ainda. A chamada a esse processo faz com que o controlador solicite a apresentação do formulário de login na página da PizzaNet.

Pode acontecer do cliente não estar cadastrado no sistema, nesse caso ele tem a opção de se autorregar, conforme é demonstrado pelo fragmento combinado do tipo **opt**, que envolve o comportamento que será executado caso o cliente não esteja registrado ainda no sistema, ou seja, o processo de **Autorregar**, que será chamado pelo uso de interação que referencia esse processo.

Se já estiver registrado, o cliente informará então seu login e senha, fazendo com que o controlador dispare o método **login**. O retorno do valor **verdadeiro** para o controlador fará com que este ordene a apresentação da mensagem “Cliente autenticado”.

Diagrama de Sequência Autorregar

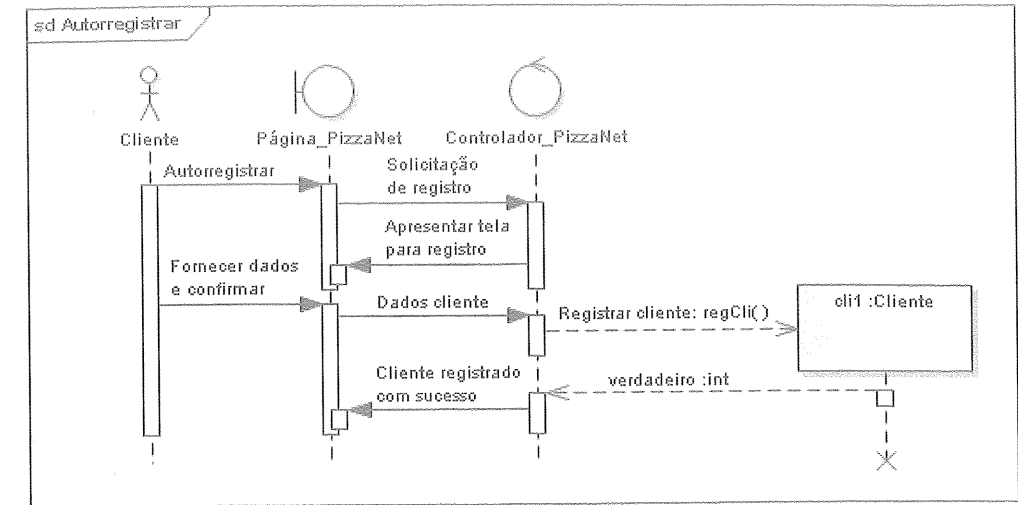


Figura 16.11 – Diagrama de Sequência Autorregar.

Este é um processo bastante simples. No momento em que o cliente solicita a execução desse processo, o controlador, em resposta, solicita a apresentação do formulário de registro, onde o cliente deverá inserir os seus dados e confirmar. Ao receber a confirmação, o controlador irá disparar o método **regCli** e instanciará um novo objeto da classe **Cliente**.

Diagrama de Sequência Opinar

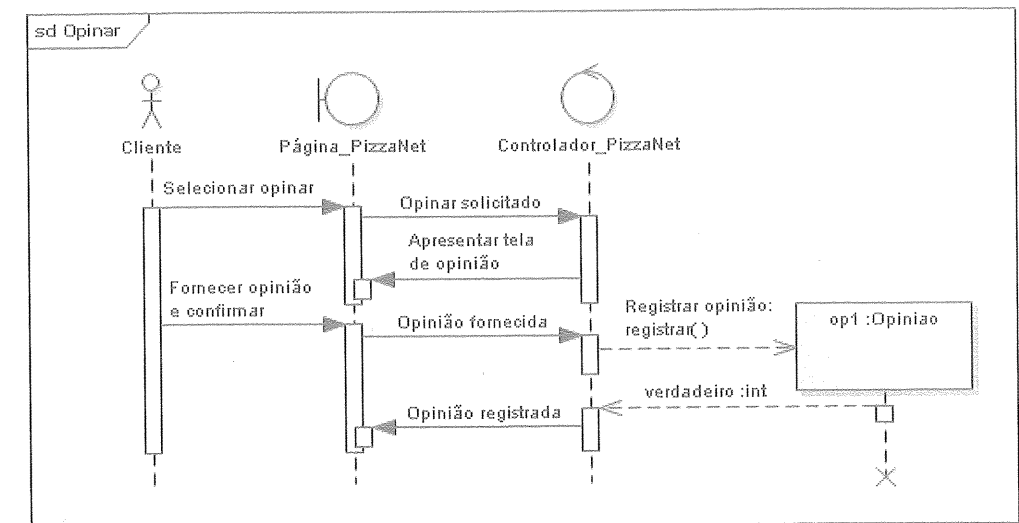


Figura 16.12 – Diagrama de Sequência Opinar.

Este é também um processo muito simples, onde, ao receber a solicitação de execução dessa funcionalidade, o controlador ordena que seja apresentado o formulário para registro de opiniões. Ao receber a opinião do cliente, o controlador executará o método registrar, para instanciar um novo objeto da classe Opinião e o processo estará finalizado.

Diagrama de Sequência Visualizar Pedido

Quando o cliente solicita a visualização dos itens de seu pedido, o controlador dispara o método conPed em um objeto da classe Pedido, para retornar as informações do pedido feito pelo cliente, ou seja, as pizzas e bebidas solicitadas.

Como os objetos da classe Pedido precisam que suas informações sejam complementadas por objetos da classe Pizza e Item_Bebida, o método conPed dispara o método conPizza para consultar cada pizza associada ao pedido, como demonstra o fragmento combinado do tipo loop. Esse método, por sua vez, dispara o método conDesTam em um objeto da classe Tamanho, para retornar o tamanho da pizza consultada e, em seguida chama o método conItemSabor em cada objeto da classe Item_Sabor associado à pizza, como demonstra o segundo fragmento combinado do tipo loop, uma vez que um objeto da classe Pizza também precisa que suas informações sejam complementadas pelos objetos dessa classe. Ainda dentro desse laço, o método conItemSabor chama o método conDesSabor para retornar a descrição do sabor relacionado ao objeto Item_Sabor em cada iteração do laço. Os resultados da consulta de cada pizza são retornados na forma de uma String para o método conPed.

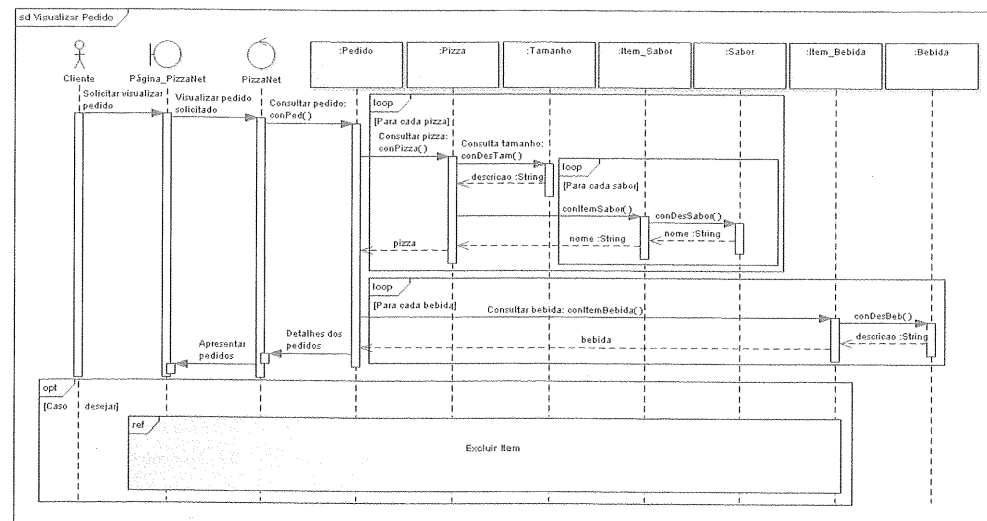


Figura 16.13 – Diagrama de Sequência Visualizar Pedido.

O método conPed então inicia um segundo laço, demonstrado por mais um fragmento combinado do tipo loop, disparando o método conItemBebida em cada objeto da classe Item_Bebida relacionado ao pedido, e esse método, por sua vez, chama o método conDesBeb para retornar a descrição da bebida. As informações de cada bebida são então retornadas ao método conPed que, de posse de todas essas informações, retorna-as ao controlador que as manda apresentar na página da PizzaNet.

A partir dessa listagem, o cliente pode excluir qualquer um dos itens apresentados. Como isso é opcional, esse comportamento é apresentado por meio de um fragmento combinado do tipo opt, que contém um uso de interação que chama o processo de Excluir Item.

Diagrama de Sequência Excluir Item

Esse processo é chamado a partir do processo anterior, e quando se inicia, o controlador precisa escolher entre dois comportamentos possíveis, como demonstra o fragmento combinado do tipo alt.

Assim, se o item a excluir for uma pizza, o controlador dispara o método excPizza para excluir a pizza em questão. Esse método, por sua vez, dispara o método excItemSabor em cada objeto da classe Item_Sabor associado à pizza. Isso é necessário porque existe uma associação de composição entre a classe Pizza e a classe Item_Sabor e por esse motivo, quando um objeto da classe Pizza for excluído, devem ser excluídos também todos os objetos Item_Sabor a ele associados. Observe que esses dois métodos são métodos destrutores, como demonstra o “x” que interrompe a linha de vida dos objetos.

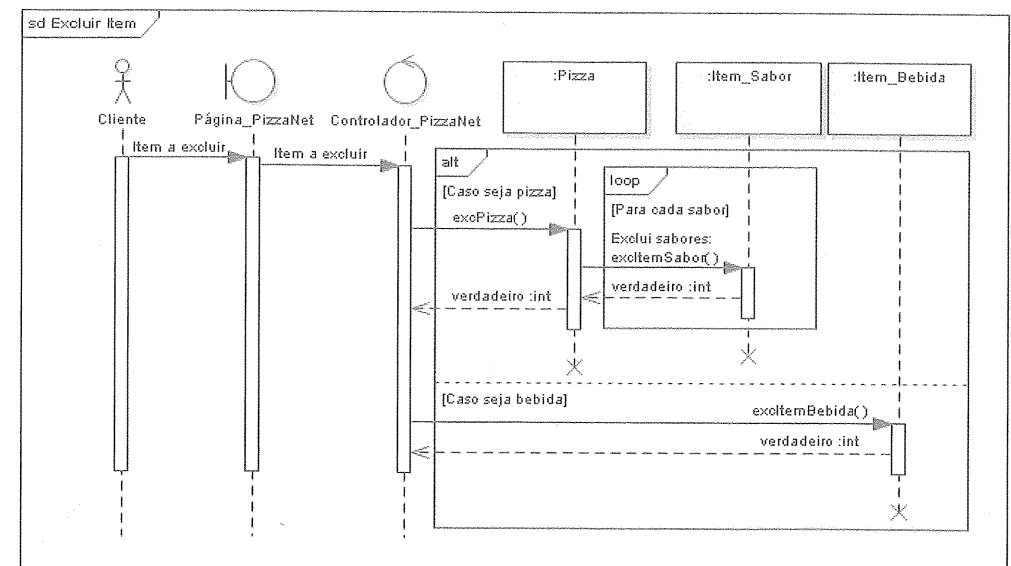


Figura 16.14 – Diagrama de Sequência Excluir Item.

Já se o item a excluir for uma bebida, o controlador chama o método `excItemBebida` para destruir o objeto da classe `Item_Bebida`. Este também é um método destrutor, como também demonstra o “X” que interrompe a linha de vida do objeto da classe `Item_Bebida`.

Diagrama de Sequência Visualizar Pedidos Anteriores

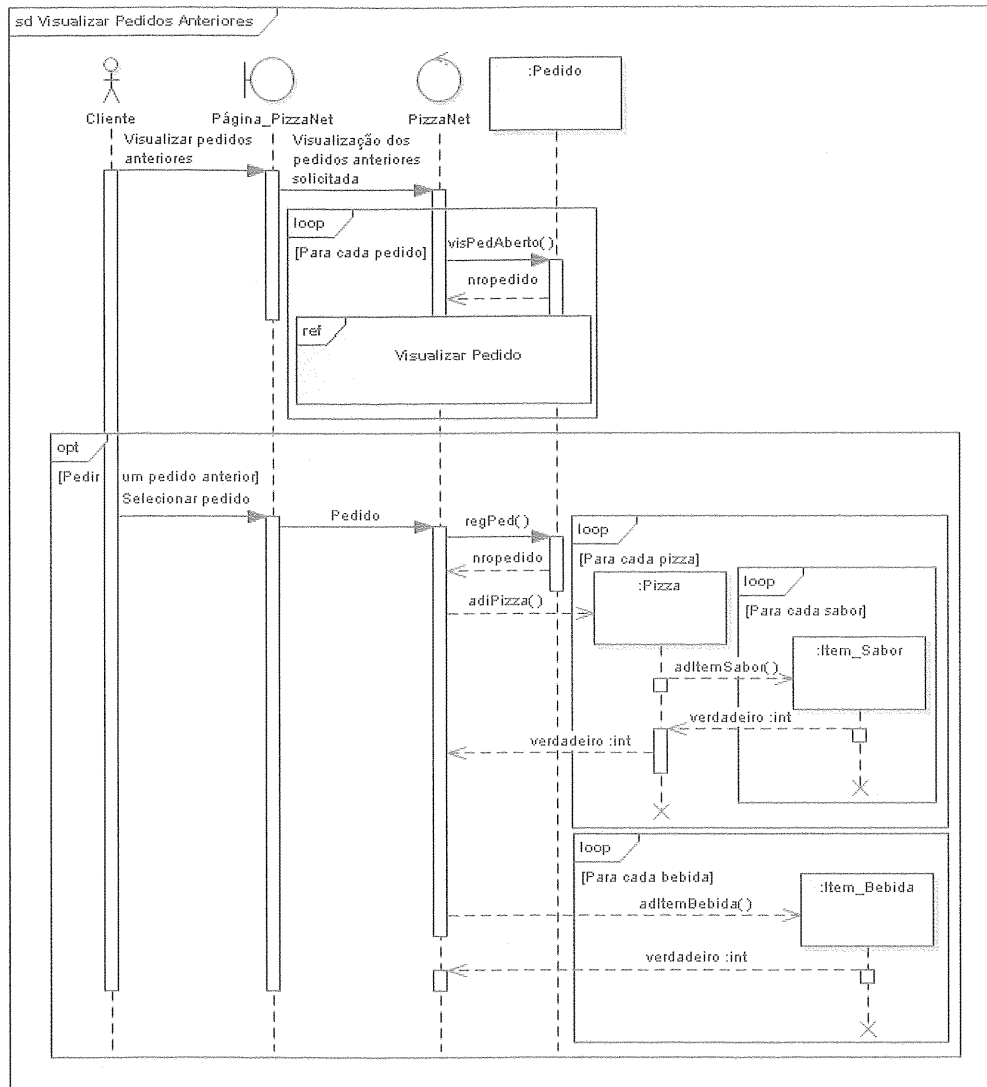


Figura 16.15 – Diagrama de Sequência Visualizar Pedidos Anteriores.

Essa funcionalidade só pode ser solicitada se o cliente já estiver autenticado. Quando o cliente seleciona essa opção, o controlador entra em um laço, demonstrado por um fragmento combinado do tipo `loop`, em que disparará o método

`visPedAberto` para cada objeto da classe `Pedido` associado ao cliente. Esse método retornará o número de um pedido e, com esse número, chama-se o processo `Visualizar Pedido` aqui referenciado por um uso de interação, que apresentará os detalhes do pedido.

A partir da listagem dos pedidos anteriores, o cliente tem a opção de solicitar um pedido novamente. Como isso é opcional, esse comportamento é representado por um fragmento combinado do tipo `opt`. Nesse caso, o cliente deverá selecionar um pedido, o que fará com que o controlador dispare o método `regPed` para instanciar um novo pedido.

Após o obter o número do novo pedido, o controlador dispara o método `adiPizza` para instanciar um novo objeto da classe `Pizza` para cada pizza contida no pedido anterior escolhido, como demonstra o fragmento combinado do tipo `loop`. Esse método por sua vez inicia um novo laço, representado por um segundo fragmento combinado do tipo `loop`, disparando o método `adItemSabor` para instanciar um novo objeto da classe `Item_Sabor` para cada sabor da pizza contida no pedido anterior.

Após instanciar as pizzas do novo pedido, o controlador inicia um novo laço, representado por mais um fragmento combinado do tipo `loop`, onde é disparado o método `adItemBebida`, instanciando um objeto da classe `Item_Bebida`, para cada bebida contida no pedido anterior.

Diagrama de Sequência Visualizar Sabores Mais Pedidos

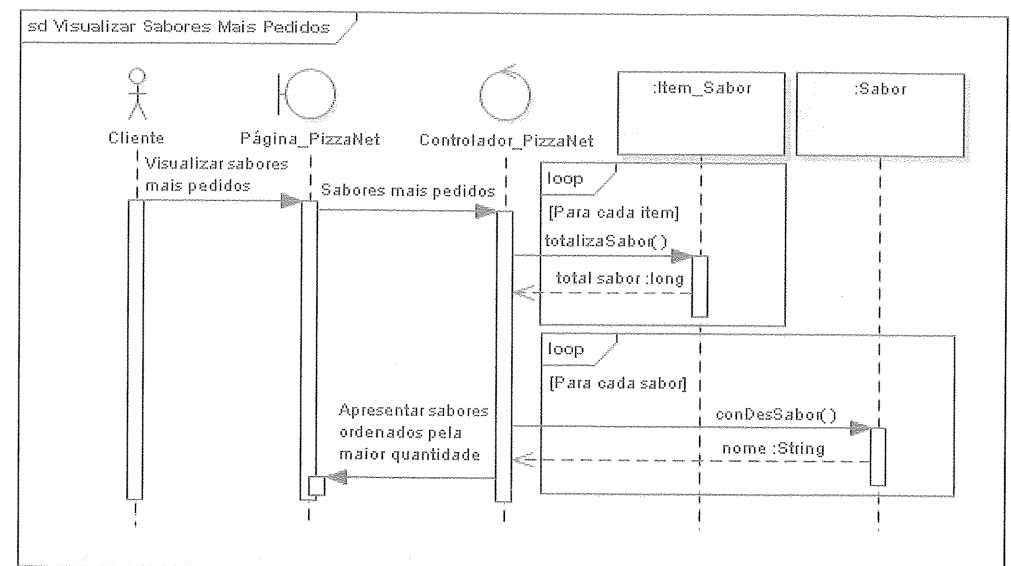


Figura 16.16 – Diagrama de Sequência Visualizar Sabores Mais Pedidos.

Nesse processo, o controlador executa um laço, representado por um fragmento combinado do tipo *loop*, onde é chamado o método *totalizaSabor*, esse método soma todos os objetos da classe *Item_Sabor* associados a um determinado sabor, retornando seu total quando o sabor muda e reiniciando a contagem.

Ao receber esses totais, o controlador inicia outro laço para consultar a descrição de cada sabor totalizado, por meio do método *conDesSabor*, que retorna uma *String* contendo a descrição do sabor consultado.

Diagrama de Sequência Concluir Pedido

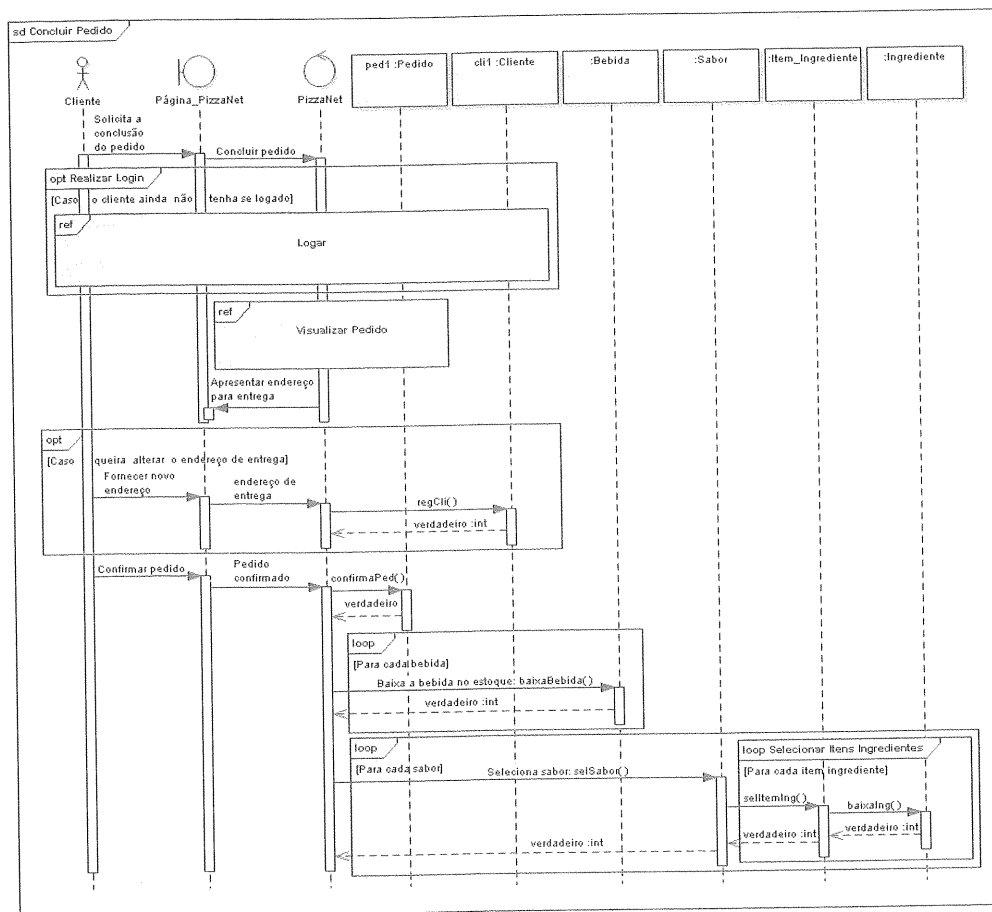


Figura 16.17 – Diagrama de Sequência Concluir Pedido.

Ao receber a solicitação de conclusão do pedido por parte do cliente, o controlador primeiramente necessita verificar se o cliente já se encontra autenticado, conforme demonstra o fragmento combinado do tipo *opt*. Caso o cliente ainda não tenha se logado, é necessário executar o processo de *Logar*, referenciado por meio do uso de interação contida dentro do fragmento.

A seguir, é necessário executar o processo de *Visualizar Pedido*, conforme demonstra a ocorrência de interação que o referencia, para ter certeza que é isso mesmo que o cliente deseja pedir. A seguir, o controlador ordena à página que apresente o endereço de entrega do cliente.

Em seguida, é preciso realizar outro teste, demonstrado pelo fragmento combinado do tipo *opt*, onde se deve verificar se o cliente achou necessário modificar o endereço de entrega. Se o cliente assim o desejar, deverá informar o novo endereço e confirmar. Quando isso ocorrer, o controlador irá disparar o método *regCli* para atualizar os dados do cliente.

Depois disso, quando o cliente confirmar o pedido pela última vez, o controlador disparará o método *confirmaPed* que definirá o pedido como confirmado, alterando o valor da situação do pedido para 1, o que significa que o cliente confirmou o pedido e este deve ser atendido.

A seguir, o controlador inicia um laço, representado por um fragmento combinado do tipo *loop*, onde a quantidade solicitada de cada bebida será diminuída do estoque da bebida em questão, por meio do disparo do método *baixaBebida*.

Ao terminar essa operação, o controlador inicia um novo laço, disparando o método *selSabor* para cada sabor solicitado no pedido. Esse método, por sua vez, executa o método *selItemIng*, selecionando cada um dos objetos da classe *Item_Ingrediente* associados ao objeto da classe *Sabor* selecionado. Finalmente, para cada objeto da classe *Item_Ingrediente* selecionado, o método *selItemIng* disparará o método *baixaIng* que diminuirá a quantidade definida no objeto da classe *Item_Ingrediente* do objeto da classe *Ingrediente* a ele associado.

Diagrama de Sequência Visualizar Pedidos em Aberto

Este é um processo disparado pelo funcionário, uma vez que a partir de agora iniciaremos a modelar os processos do subsistema administrativo. Quando o funcionário solicita esse serviço, o controlador inicia um laço, representado por um fragmento combinado do tipo *loop*, onde, para cada pedido em aberto, será disparado o método *visPedAberto*. Esse método retornará o número de cada pedido em aberto e, de posse dele, o controlador solicitará a execução do processo *Visualizar Pedido*, já descrito anteriormente, que apresentará os detalhes do pedido, como demonstra o uso de interação de mesmo nome.

A seguir, o funcionário pode, opcionalmente, finalizar um pedido em aberto, como demonstra o fragmento combinado do tipo *opt*. Nessa situação, o funcionário deverá selecionar o pedido em questão e solicitar sua finalização. Isso fará com que o controlador ordene a execução do processo *Finalizar Pedido Cliente*, conforme demonstra o uso de interação. Esse novo processo será descrito na próxima seção.

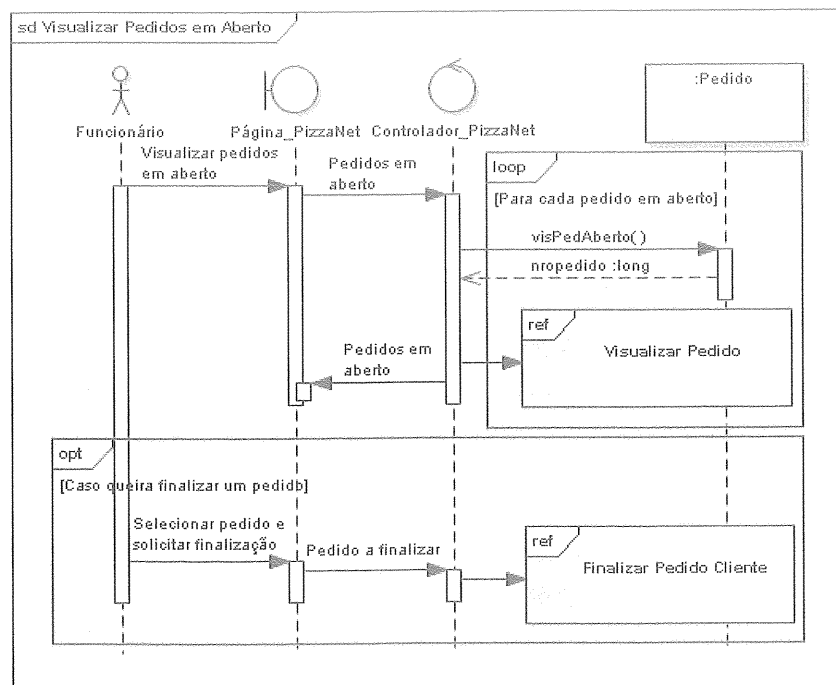


Figura 16.18 – Diagrama de Sequência Visualizar Pedidos em Aberto.

Diagrama de Sequência Finalizar Pedido Cliente

Devido ao fato desse processo ser chamado a partir do processo de Visualizar Pedidos em Aberto, ele se inicia com o controlador executando um laço, como demonstra o fragmento combinado do tipo **loop**, onde é disparado o método **conFun** para consultar todos os funcionários da empresa, retornando seus nomes para serem apresentados no formulário de finalização de pedido.

Por meio desse formulário, o funcionário deve informar os funcionários que prepararam e entregaram o pedido, fazendo com que o controlador dispare o método **finalizaPed** em um objeto da classe **Pedido**, definindo-o como finalizado, por meio da mudança do valor da situação do pedido para 2.

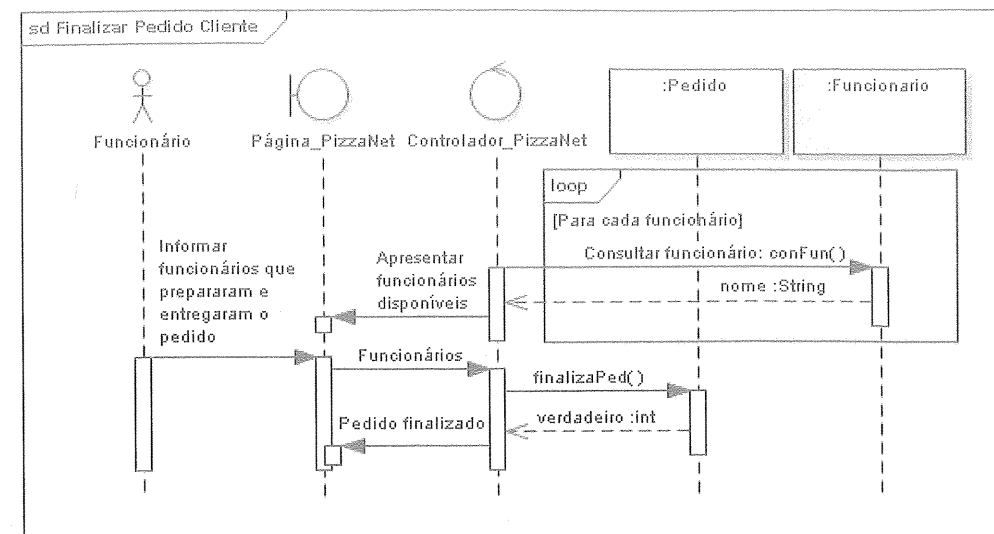


Figura 16.19 – Diagrama de Sequência Finalizar Pedido Cliente.

Diagrama de Sequência Manter Cardápio

Esta é uma funcionalidade que só pode ser solicitada pelo administrador. Quando este a solicita, o controlador em resposta inicia um laço, demonstrado pelo fragmento combinado do tipo **loop**, onde é disparado o método **conDesSabor** para recuperar a descrição de cada sabor registrado no sistema.

A seguir, o controlador inicia um segundo laço, onde dispara o método **conIng** para retornar a descrição de cada ingrediente cadastrado. Com essas informações, o controlador solicita à interface que apresenta o formulário de manutenção de cardápio ao administrador.

A partir desse formulário, o administrador precisa escolher entre incluir um novo sabor ou consultar um sabor já existente, como demonstra o fragmento combinado do tipo **alt**.

Na primeira alternativa, o administrador deverá informar o novo sabor e os ingredientes necessários à sua produção, o que fará com que o controlador dispare o método **regSab**, instanciando um novo objeto da classe **Sabor**. Embora o correto fosse inserir um novo objeto da classe **Sabor** e este só passa a existir a partir desse momento, optamos por aproveitar o objeto anterior dessa classe, para não deixar o diagrama largo demais.

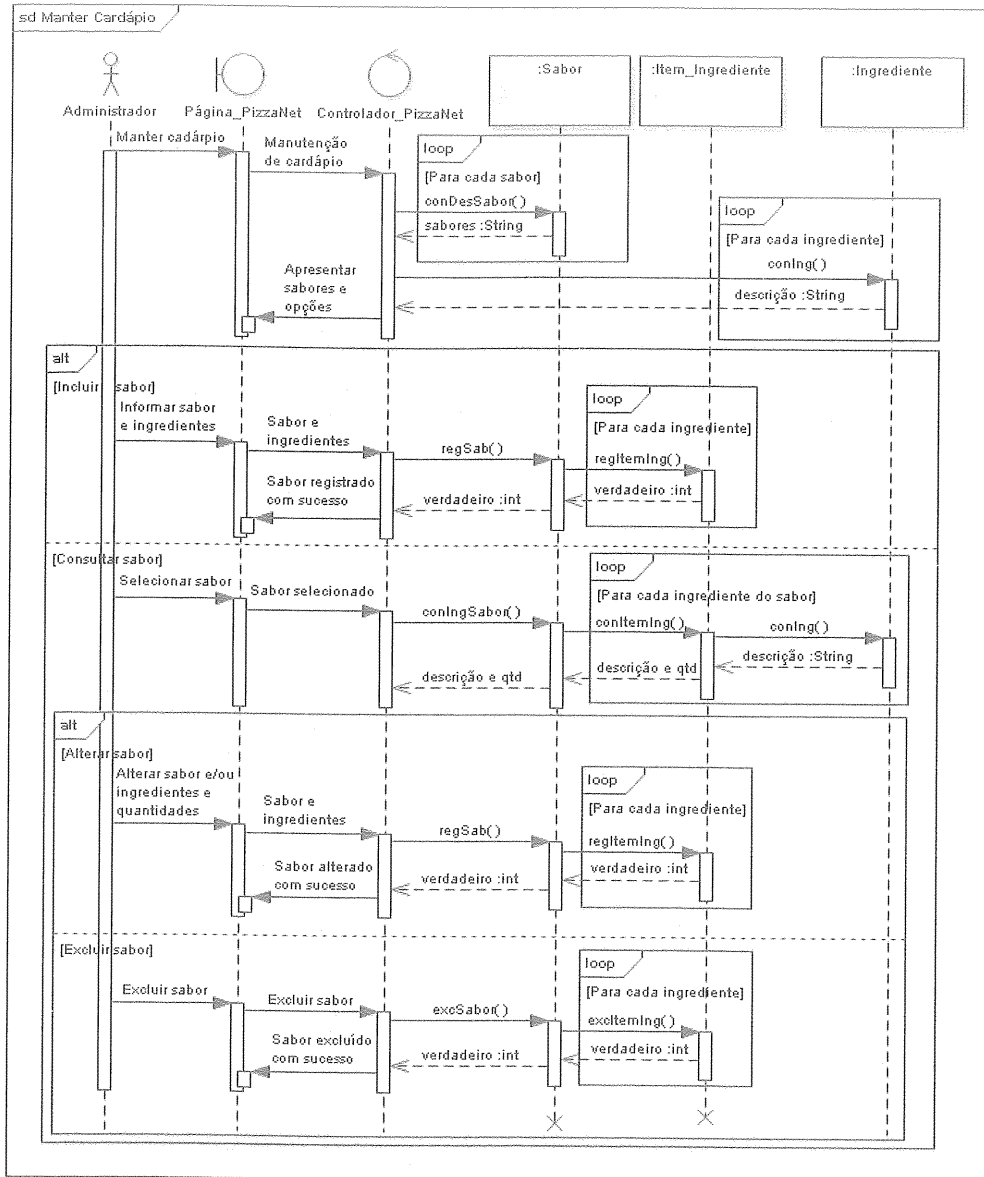


Figura 16.20 – Diagrama de Sequência Manter Cardápio.

O método `regSab`, por sua vez, inicia um loop, onde dispara o método `regItemIng`, instanciado um novo objeto da classe `Item_Ingrediente` para cada ingrediente necessário à produção do sabor. Se o retorno dos métodos for verdadeiro, o controlador mandará a interface apresentar a mensagem de **Sabor registrado com sucesso**.

Na segunda alternativa, o administrador deverá selecionar um dos sabores listados no formulário. Quando isso ocorre, o controlador dispara o método `conIngSabor` em um objeto da classe `Sabor` para consultar os ingredientes do sabor selecionado. Para que esse método possa realizar essa tarefa, ele deve iniciar um laço, onde dispara o método `conItemIng` em todo objeto da classe `Item_Ingrediente` associado ao objeto `Sabor` em questão. Esse método, por sua vez, chama o método `conIng` para retornar a descrição do objeto da classe `Ingrediente` associado a cada objeto `Item_Ingrediente`. O retorno desses métodos é enviado ao controlador, que os manda apresentar na interface.

A partir dessas informações, o administrador poderá decidir entre alterar um sabor ou excluir um sabor, como demonstra o fragmento combinado do tipo `alt`.

Caso queira alterar a descrição de um sabor ou a quantidade de seus ingredientes, o administrador deverá executar as alterações sobre o formulário e confirmar. Isso fará com que o controlador execute o mesmo procedimento para registrar um novo sabor, disparando o método `regSab` em um objeto da classe `Sabor` e o método `regItemIng` em objetos da classe `Item_Ingrediente` para cada ingrediente do sabor.

Caso o administrador deseje excluir um sabor, o controlador irá disparar o método `excSabor` no objeto da classe `Sabor` a ser destruído. Antes de finalizar, esse método executará o método `excItemIng` para destruir todos os objetos da classe `Item_Ingrediente` associados ao objeto da classe `Sabor`. Isso é necessário porque a classe `Sabor` tem uma associação de composição com a classe `Item_Ingrediente`, sendo assim, quando um objeto `Sabor` for destruído, todos os objetos `Item_Ingrediente` a ele associados devem ser também destruídos.

Diagrama de Sequência Emitir Produtos em Baixa no Estoque

Quando o administrador solicita esse relatório, o controlador inicia dois laços, como demonstram os fragmentos combinados do tipo `loop`.

No primeiro laço, o controlador dispara o método `conIngBaixa` em objetos da classe `Ingrediente` para retornar todos os ingredientes com a quantidade em estoque abaixo da quantidade mínima.

No segundo laço, o controlador dispara o método `conBebBaixa` em objetos da classe `Bebida` para retornar todas as bebidas com a quantidade em estoque abaixo do mínimo.

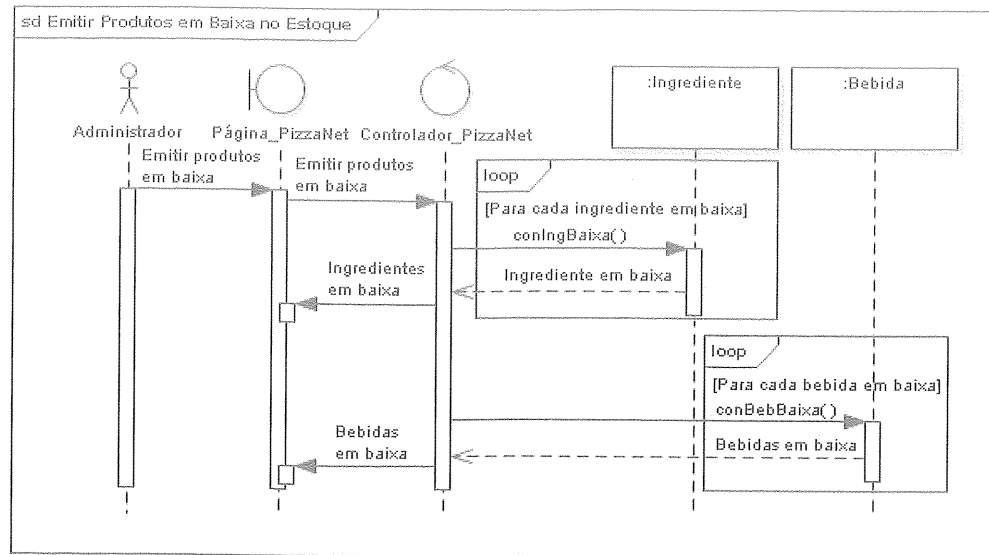


Figura 16.21 – Diagrama de Sequência Emitir Produtos em Baixa no Estoque.

Diagrama de Sequência Emitir Compras em Aberto

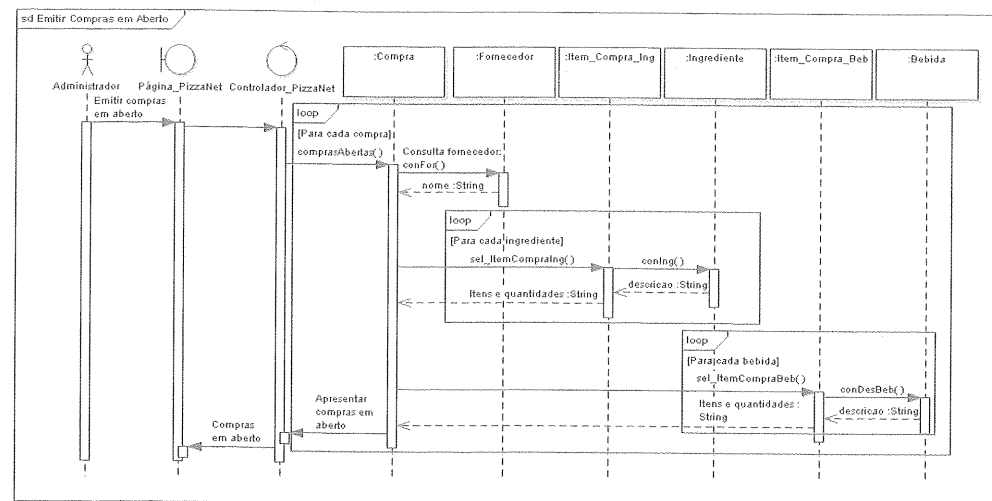


Figura 16.22 – Diagrama de Sequência Emitir Compras em Aberto.

Quando esse relatório é solicitado, o controlador inicia um grande laço, representado por um fragmento combinado do tipo `loop`, onde é disparado o método `comprasAbertas` em cada objeto da classe `Compra`, cuja situação esteja em aberto. É necessário saber para qual fornecedor foi solicitada a compra, para isso o método `comprasAbertas` dispara o método `conFor` em um objeto da classe `Fornecedor`, para retornar seus dados.

Em seguida, o método `comprasAbertas` inicia um novo laço, com o objetivo de selecionar todos os ingredientes solicitados na compra. Para isso, é necessário disparar o método `sel_ItemCompraIng` em objetos da classe `Item_Compra_Ing` e este, para recuperar a descrição de cada ingrediente, dispara o método `conIng` em um objeto da classe `Ingrediente`.

Depois disso, o método `comprasAbertas` inicia ainda um último laço, com o objetivo de selecionar todas as possíveis bebidas solicitadas na compra. Para isso é chamado o método `sel_ItemCompraBeb` em objetos da classe `Item_Compra_Beb`. Esse método, por sua vez, executa o método `conDesBeb` para recuperar a descrição de cada bebida solicitada na compra.

Com base no retorno desses métodos, o controlador solicita que essas informações sejam apresentadas na interface da PizzaNet.

Diagrama de Sequência Manter Compras Fornecedor

Quando o administrador solicita a execução do processo de manutenção de compras, o controlador dispara a execução do processo de emissão de compras em aberto, descrito na seção anterior, como demonstra a ocorrência de interação que refere-se a esse processo. Com base nos resultados desse processo, o controlador solicita a apresentação do formulário de manutenção de compras, já carregado com as compras em aberto.

Nesse formulário, o administrador pode escolher entre finalizar uma compra ou registrar uma compra nova, como demonstra o fragmento combinado do tipo `alt`.

Na primeira alternativa, o administrador deve selecionar a compra que deseja finalizar e marcá-la como concluída, o que faz com que o controlador dispare o método `fecharCompra` em um objeto da classe `Compra`, modificando o estado de sua situação para 1, que significa que a compra está fechada.

Se a segunda alternativa for escolhida, o controlador primeiramente executará três laços, representados por fragmentos combinados do tipo `loop`, para prover informações ao formulário de manutenção de compras. No primeiro laço ele disparará o método `conIng` para retornar a descrição de todos os ingredientes registrados. No segundo laço o controlador chamará o método `conDesBeb` para retornar a descrição de cada bebida cadastrada. Finalmente, no terceiro laço, o controlador executará o método `conFor`, para retornar o nome de todos os fornecedores da empresa.

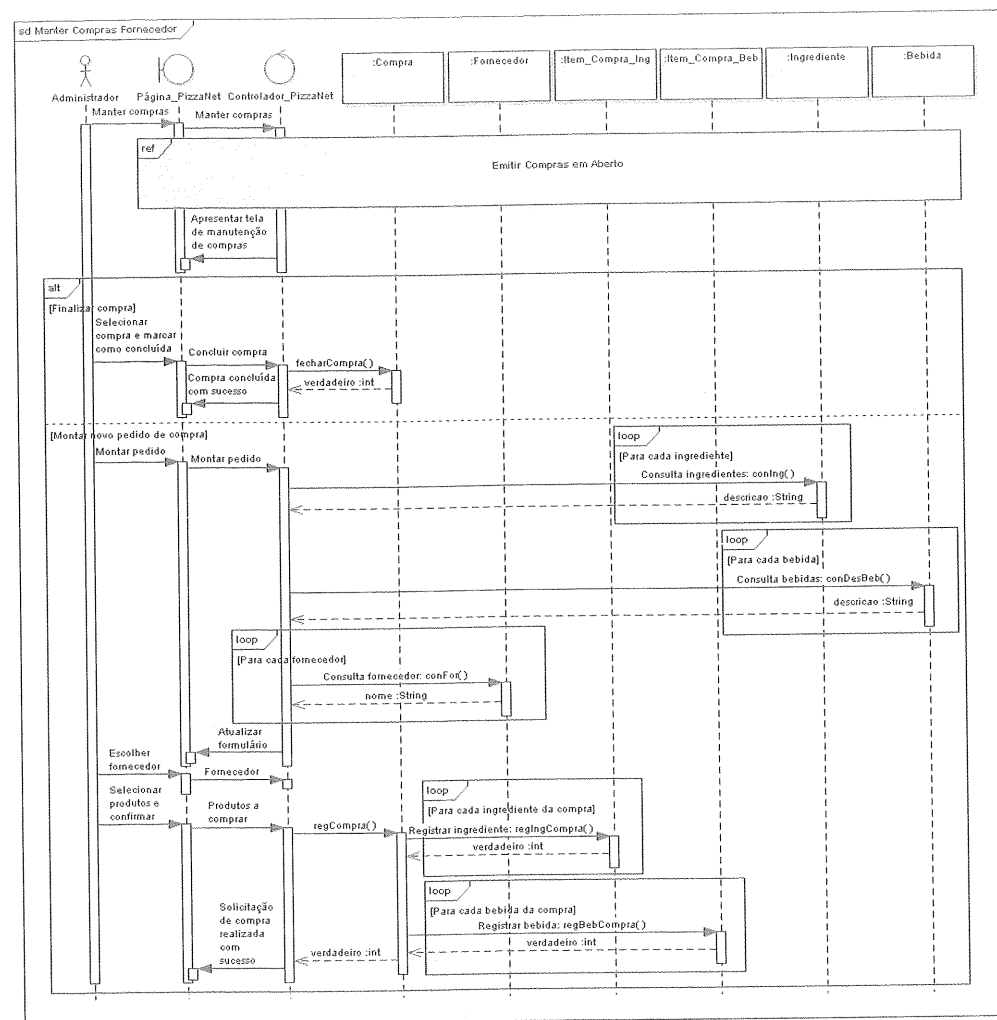


Figura 16.23 – Diagrama de Sequência Manter Compras Fornecedor.

A seguir, o administrador poderá escolher o fornecedor para o qual deseja solicitar produtos e, em seguida, poderá selecionar os produtos que deseja comprar. Isso fará com que o controlador dispare o método `regCompra`, gerando um novo objeto da classe `Compra`.

Esse método, por sua vez, inicia um laço onde chamará o método `regIngCompra` para instanciar um novo objeto da classe `Item_Compra_Ing` para cada possível ingrediente que o administrador deseja solicitar. Após a finalização desse laço, é iniciado um segundo laço, onde será executado o método `regBebCompra` para instanciar um novo objeto da classe `Item_Compra_Beb` para cada possível bebida solicitada na compra.

Se o retorno desses métodos for verdadeiro, o controlador solicitará que a interface apresente uma mensagem informando que a solicitação de compra foi realizada com sucesso.

Diagrama de Sequência Emitir Melhores Clientes

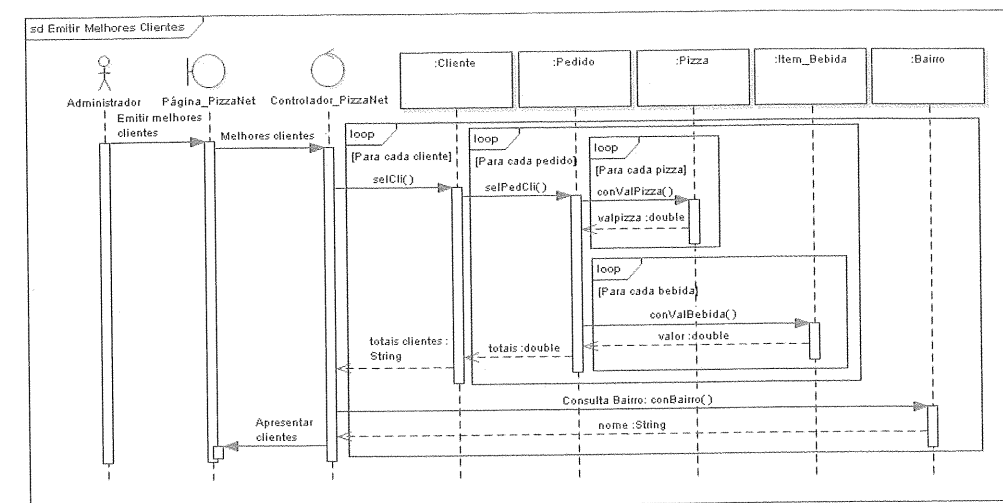


Figura 16.24 – Diagrama de Sequência Emitir Melhores Clientes.

Quando esse relatório é solicitado pelo administrador, o controlador inicia um grande laço, representado por um fragmento combinado do tipo `loop`, onde, para cada cliente, é disparado o método `seICli`, que seleciona os clientes da empresa. Esse método, por sua vez, inicia um novo laço, disparando o método `seIPedCli` para cada pedido feito por um cliente específico. O método `seIPedCli` em resposta inicia dois laços, onde, no primeiro é disparado o método `conValPizza`, para retornar o valor de cada possível pizza do pedido. Já no segundo laço é chamado o método `conValBebida`, para retornar o valor de cada possível bebida do pedido. O método `seIPedCli` totaliza os valores de cada pedido e os retorna ao método `seICli`, que totaliza todos os pedidos de um cliente específico e os retorna ao controlador.

Após obter os totais de um cliente específico, o controlador consulta o bairro onde mora o cliente, por meio do método `conBairro`. Sendo assim possível saber os bairros dos melhores clientes da pizzaria. Esse processo é executado enquanto houver clientes registrados no sistema.

Diagrama de Sequência Emitir Consumo por Período

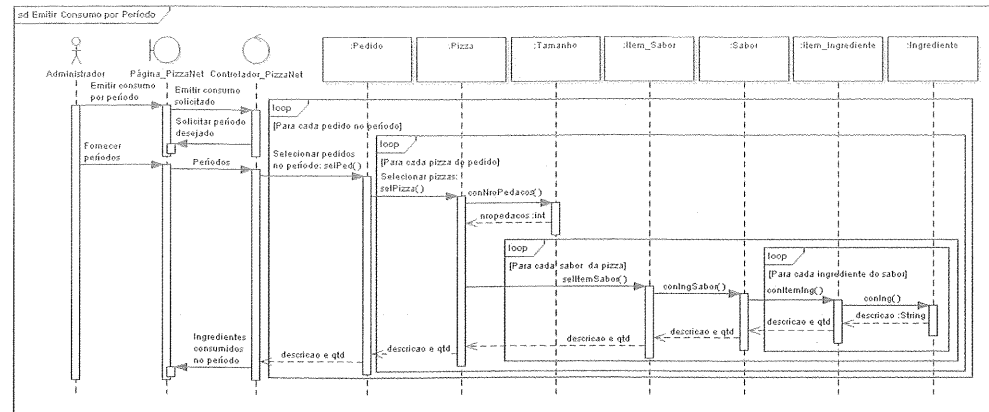


Figura 16.25 – Diagrama de Sequência Emitir Consumo por Período.

A solicitação desse relatório faz com que o controlador peça à interface que solicite ao administrador os períodos iniciais e finais da listagem. O administrador deve então fornecer os períodos desejados. Isso faz com que o controlador inicie um grande laço, representado por um fragmento combinado do tipo loop, onde, para cada objeto da classe Pedido será disparado o método selPed, para selecionar os pedidos que estejam dentro do período informado.

Sempre que um pedido for encontrado dentro do período informado, esse método inicia outro grande laço, para selecionar cada pizza do período, por meio da execução do método selPizza sobre objetos da classe Pizza. Esse último método dispara o método conNroPedacos em um objeto da classe Tamanho, para retornar o número de pedaços de cada objeto da classe Pizza.

Em seguida, o método selPizza inicia um novo laço, disparando o método selItemSabor para selecionar todos os objetos da classe Item_Sabor associados à pizza. O método selItemSabor, por sua vez, dispara o método conIngSabor no objeto da classe Sabor associado ao objeto da classe Item_Sabor, para retornar todos os ingredientes do sabor. Para poder retornar a informação desejada, o método conIngSabor inicia um novo laço para selecionar todos os objetos da classe Item_Ingrediente associados ao objeto da classe Sabor do momento, por meio do método conItemIng. Dentro desse laço, esse último método dispara o método conIng no objeto da classe Ingrediente associado ao objeto Item_Ingrediente para retornar a descrição do ingrediente em questão.

As informações retornadas por esses métodos são manipuladas pelo controlador que as organiza e solicita sua apresentação na interface.

16.2.7 Diagrama de Comunicação Escolher Pizza

Nesta seção apenas apresentaremos o diagrama de comunicação equivalente ao diagramas de sequência Escolher Pizza, explicado na seção anterior. Optamos por apresentar somente esse processo porque esses diagramas demonstram praticamente as mesmas informações que os de sequência, mostradas de uma maneira diferente. Assim, esse diagrama tem um caráter basicamente ilustrativo de como utilizar o diagrama de comunicação para modelar o mesmo processo.

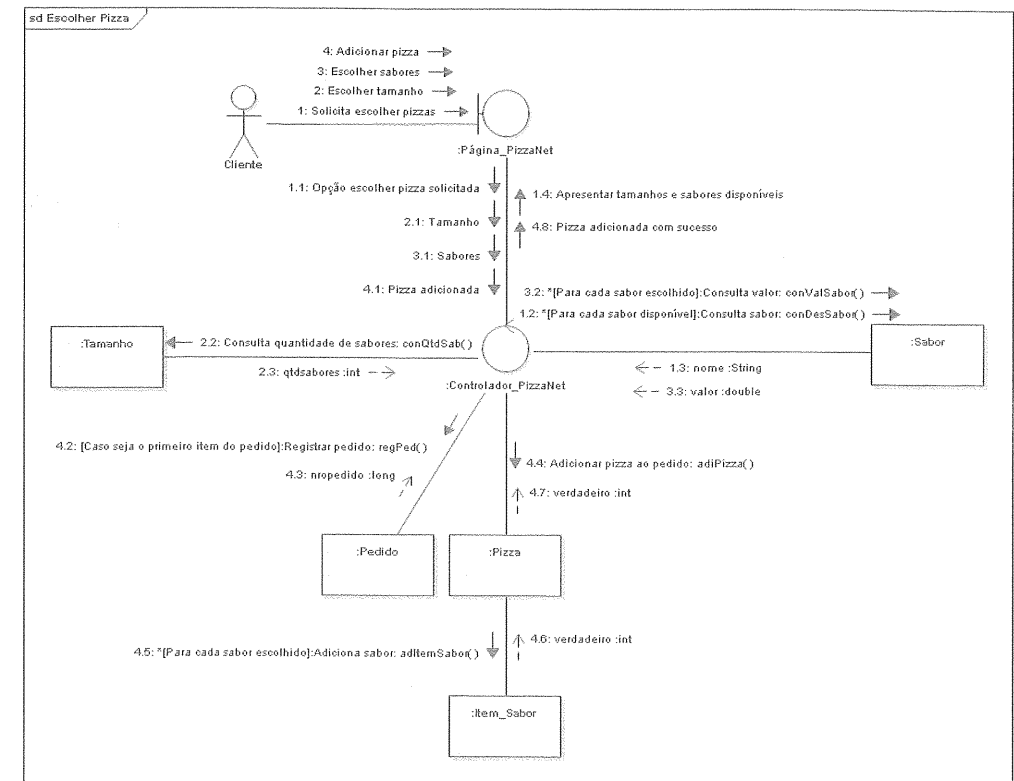


Figura 16.26 – Diagrama de Comunicação Escolher Pizza.

O leitor deve observar que os laços representados no diagrama de sequência por fragmentos combinados do tipo loop, aqui são representados pelo operador (*) e por condições de guarda. Estas são também usadas para estabelecer situações opcionais, representadas no diagrama de sequência por fragmentos combinados do tipo opt.

16.2.8 Diagramas de Máquinas de Estados da PizzaNet

Nesta seção explicaremos os diagramas de máquina de estados referentes aos processos do sistema PizzaNet. Embora esse diagrama possa ser usado para modelar os estados de um único objeto, preferimos modelar os estados dos casos de uso do sistema, pois, uma vez que o sistema que estamos modelando é um sistema comercial, o número de estados assumidos por determinados objetos em certos processos é ínfimo, por diversas vezes resumindo-se a um único estado. Assim, consideramos mais válido modelar os estados dos processos como um todo.

Diagrama de Máquina de Estados Escolher Pizza

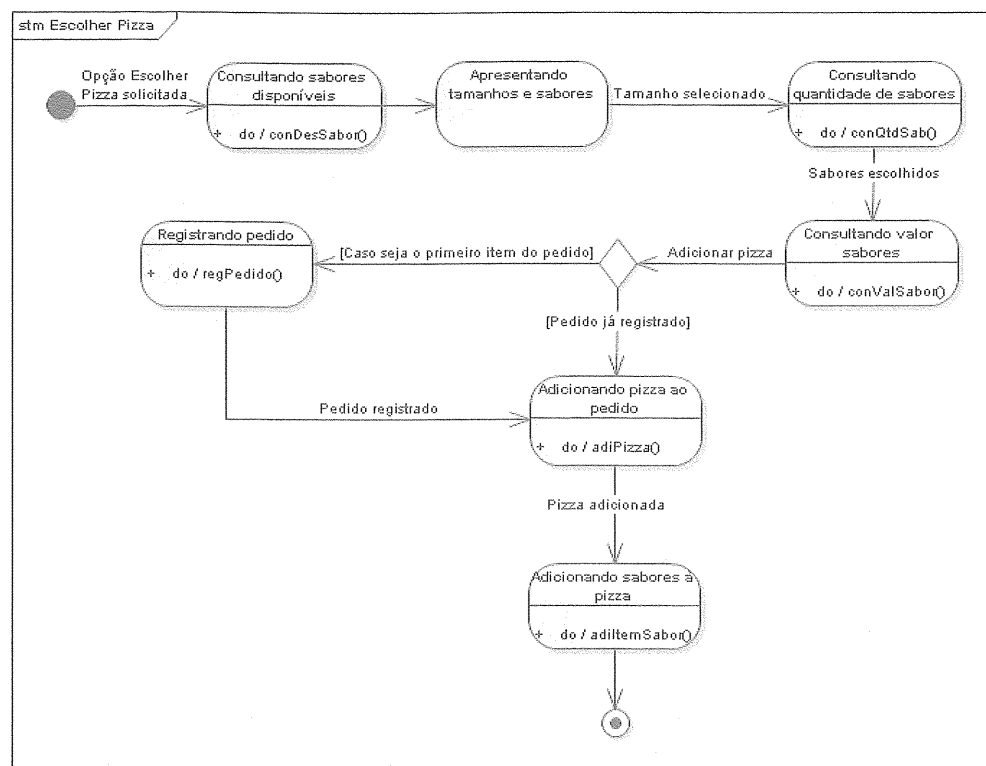


Figura 16.27 – Diagrama de Máquina de Estados Escolher Pizza.

Esse processo se inicia quando o cliente acessa a página da PizzaNet ou quando o cliente pressiona o botão Pizzas. Esse evento gera uma transição para o estado Consultando sabores disponíveis, onde é executado o método conDesSabor, conforme demonstra a cláusula Do. Após a conclusão desse estado passa-se ao estado Apresentando tamanhos e sabores, onde se espera que o cliente escolha um tamanho de pizza e seus respectivos sabores. Quando o tamanho é escolhido,

gera-se uma transição para o estado Consultando quantidade de sabores, onde é executado o método conQtdSab.

Em seguida, quando o cliente escolhe os sabores da pizza, é gerada uma transição para o estado Consultando valor sabores, onde é executado o método conValSabor, para recuperar o valor de cada sabor escolhido. Após a conclusão desse estado, o processo passa a um pseudoestado de escolha, no qual é preciso verificar se o item adicionado é o primeiro item do pedido. Se isso for verdade, gera-se uma transição para o estado Registrando pedido, no qual é disparado o método regPedido, para gerar um novo objeto da classe Pedido.

Após registrar o pedido ou se este não for o primeiro item do pedido, passa-se ao estado Adicionando pizza ao pedido, em que é chamado o método adiPizza. Quando a pizza for adicionada, gera-se o estado Adicionando Sabores à Pizza e é executado o método AdiItemSabor, para gerar um novo objeto da classe Item_Sabor para cada sabor escolhido para a pizza.

Diagrama de Máquina de Estados Escolher Bebida

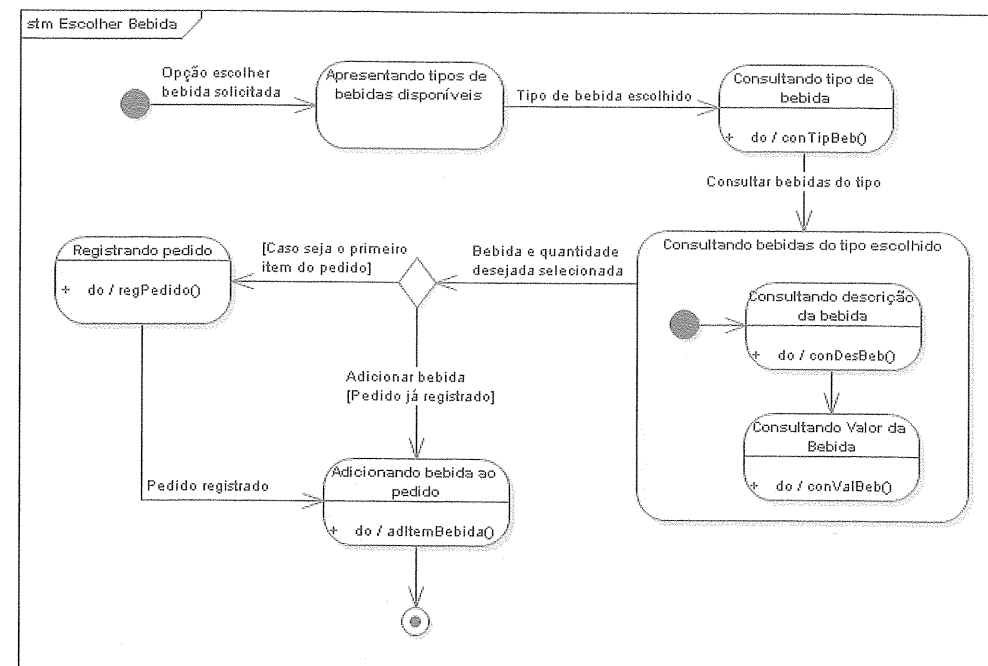


Figura 16.28 – Diagrama de Máquina de Estados Escolher Bebida.

Quando o cliente escolhe essa opção, é gerada uma transição para o estado Apresentando tipos de bebidas disponíveis, onde se espera que o cliente selecione um tipo de bebida. No momento em que o cliente escolher um tipo de bebida é

gerada uma transição para o estado Consultando tipo de bebida, onde é executado o método `conTipBeb`.

Quando esse estado se encerra, é gerada uma transição para um estado composto no qual são consultadas as bebidas do tipo escolhido. Nesse estado composto são gerados os subestados Consultando descrição da bebida e Consultando valor da bebida, executando os métodos `conDesBeb` e `conValBeb`.

No momento em que o cliente escolher a bebida e a quantidade desejada, é gerado uma transição para o um pseudoestado de escolha, onde é preciso verificar se a bebida escolhida refere-se ao primeiro item do pedido. Se esse for o caso, gera-se uma transição para o estado Registrando pedido e é executado o método `regPed`. A seguir, ou caso a bebida escolhida não seja o primeiro item do pedido, é gerada uma transição para o estado Adicionando bebida ao pedido, onde é executado o método `adItemBebida` para instanciar um novo objeto da classe `Item_Bebida` associado ao pedido em questão.

Diagrama de Máquina de Estados Logar

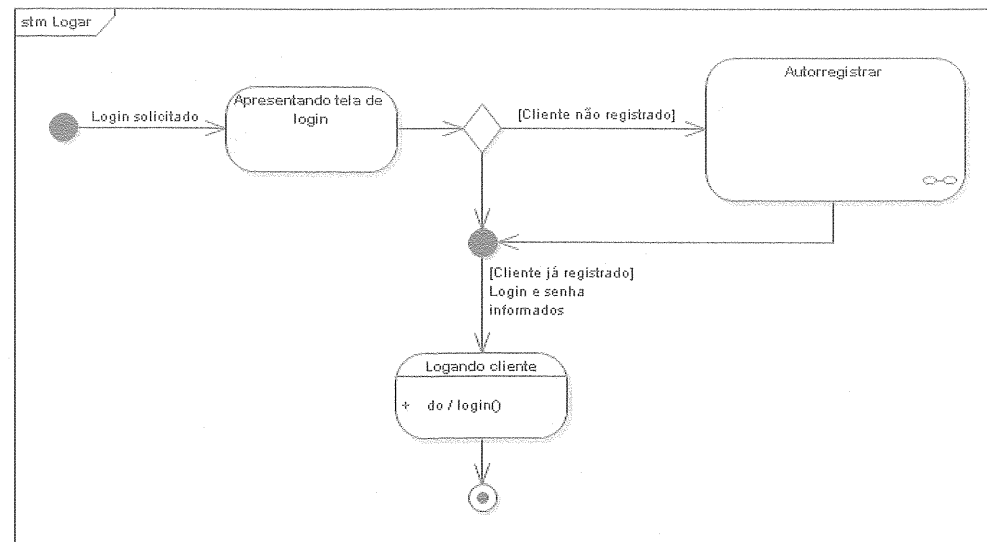


Figura 16.29 – Diagrama Máquina de Estados Logar.

No momento em que o cliente seleciona essa opção, uma transição gera o estado Apresentando tela de login, onde supõe-se que o cliente informe seu nome-login e senha. Porém pode acontecer do cliente não estar registrado no sistema, nesse caso ele poderá se autorregistrar.

Por esse motivo, existe um pseudoestado de escolha depois desse estado, onde se deve decidir entre duas situações possíveis. Caso o cliente não esteja registrado, é gerada uma transição para um estado de submáquina que se refere ao processo de Autorregistrar, descrito em outro diagrama, e, após isso, passa-se ao estado Logando cliente, onde é executado o método `login`. Caso o cliente já esteja registrado passa-se diretamente ao último estado descrito. Observe que unimos o fluxo dividido por meio de um pseudoestado de junção.

Diagrama de Máquina de Estados Autorregistrar

Esse processo envolve apenas dois estados, sendo o primeiro estado Recebendo dados do cliente, gerado pela transição causada pela solicitação de registro pelo cliente. Nesse estado o cliente deve informar seus dados e confirmar. Quando isso ocorre é gerada uma transição para o estado Registrando Cliente, onde é executado o método `regCli`, para instanciar um novo objeto da classe `Cliente`.

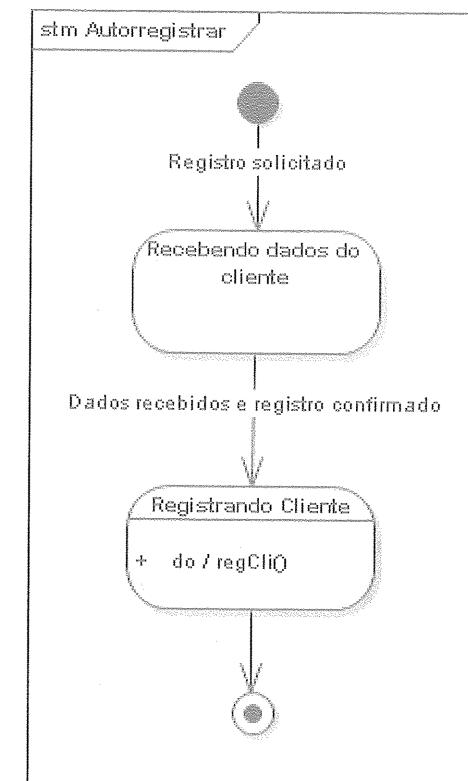


Figura 16.30 – Diagrama de Máquina de Estados Autorregistrar.

Diagrama de Máquina de Estados Opinar

Esse é um diagrama bastante simples, composto de apenas dois estados. Ele inicia-se quando o cliente solicita a execução da funcionalidade de registrar opinião. Isso causa uma transição que gera o estado **Apresentando formulário de opinião**, que espera que o cliente informe a sua opinião e confirme. Quando isso acontece é gerado o estado **Registrando opinião**, onde é executado o método registrar, que instancia um novo objeto da classe *Opinioao*.

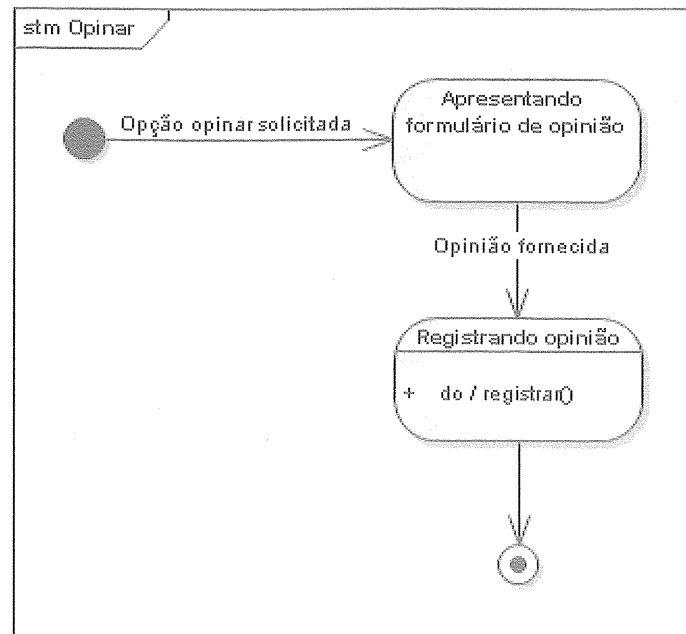


Figura 16.31 – Diagrama de Máquina de Estados Opinar.

Diagrama de Máquina de Estados Visualizar Pedido

Os estados modelados por esse diagrama são iniciados quando um cliente solicita a visualização de seu pedido. Isso produz uma transição para o estado **Consultando pedido**, em que é executado o método *conPed*. Após o pedido ter sido consultado, é gerada uma transição para um estado composto que irá consultar as possíveis pizzas do pedido. Esse estado composto contém os subestados **Consultando pizza**, que executa o método *conPizza*; **Consultando tamanho**, que dispara o método *conDesTam*; **Consultando itens de sabor**, que chama o método *conItemSabor* para carregar todos os objetos da classe *Item_Sabor* associados ao pedido; e **Consultando sabor**, que executa o método *conDesSabor*. Os subestados são gerados na ordem em que foram citados.

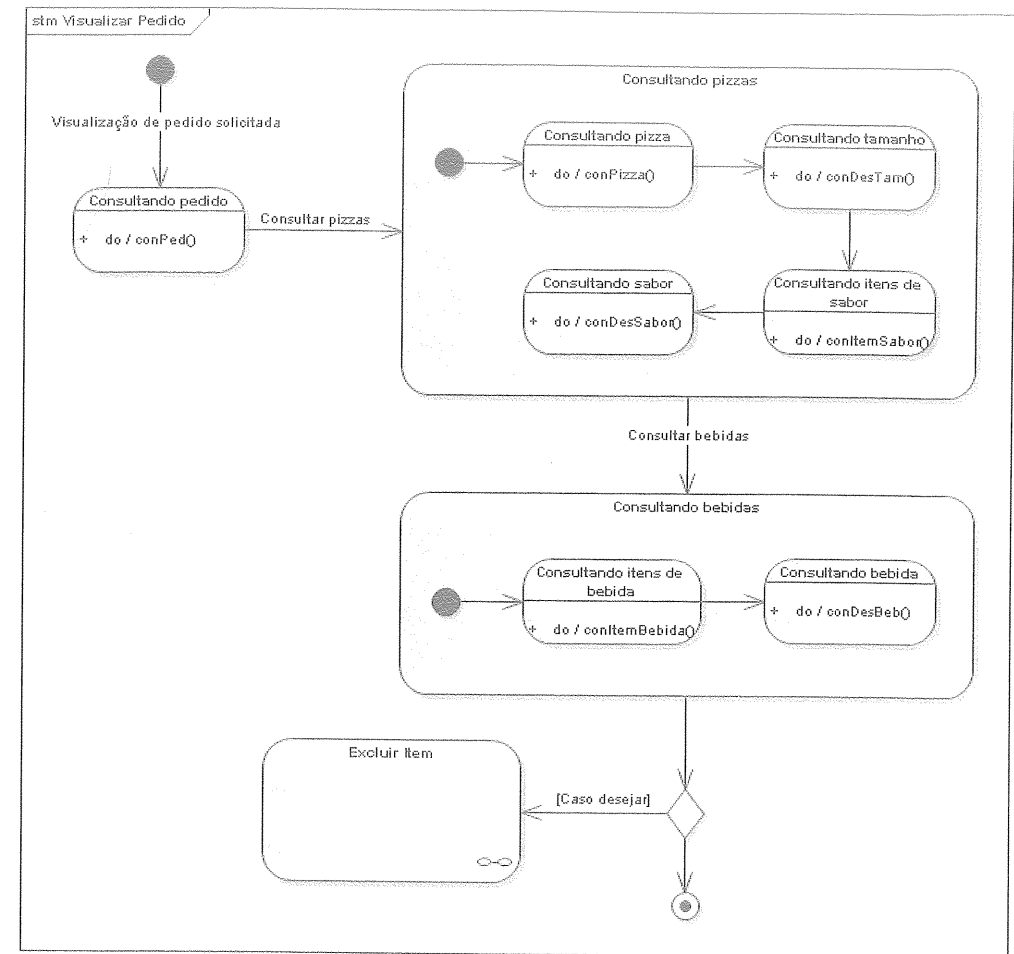


Figura 16.32 – Diagrama de Máquina de Estados Visualizar Pedido.

Após terem sido consultadas todas as possíveis pizzas solicitadas no pedido, a atividade passa a um novo estado composto, onde são pesquisadas as possíveis bebidas solicitadas no pedido. Esse estado composto contém os subestados **Consultando itens de bebida**, que executa o método *conItemBebida*, onde são carregados todos os objetos da classe *Item_Bebida* associados ao pedido em questão; e **Consultando bebida**, que chama o método *conDesBeb*.

Após ter apresentado os dados das pizzas e bebidas do pedido, o processo encontra um pseudoestado de escolha, onde é oferecida a opção ao cliente de excluir qualquer item visualizado. Se o cliente optar por excluir algum item, é gerada uma transição para o estado de submáquina **Excluir item**, que faz referência ao processo de mesmo nome, detalhado em um diagrama em separado.

Diagrama de Máquina de Estados Excluir Item

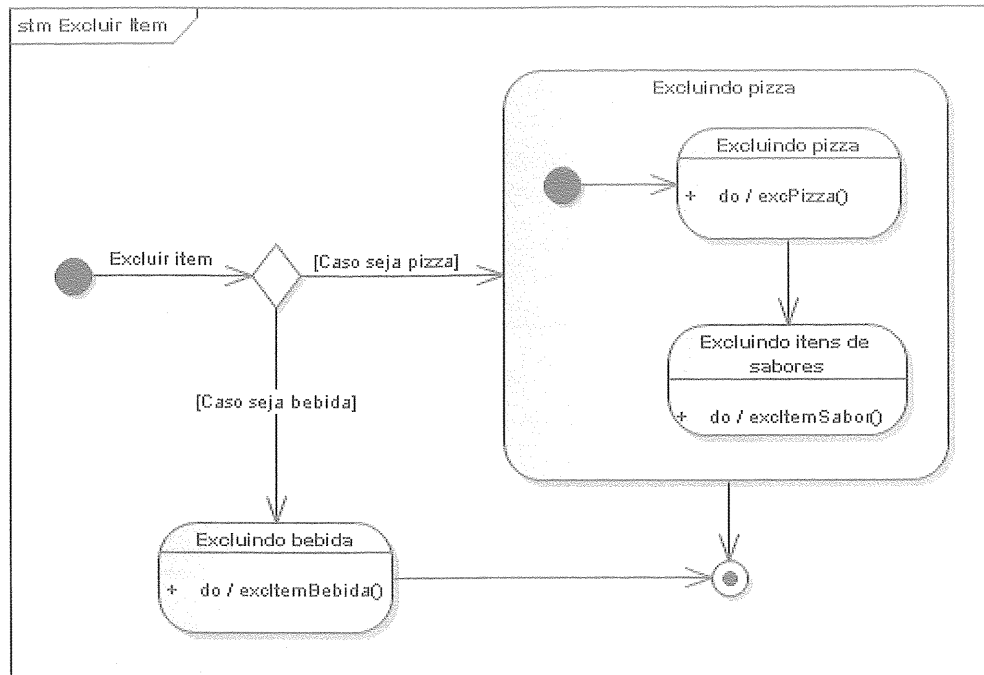


Figura 16.33 – Diagrama de Máquina de Estados Excluir Item.

Os estados desse processo iniciam-se quando o cliente opta por excluir um item de um pedido, a partir do processo de visualização de pedido. A solicitação de exclusão de um item gera uma transição para um pseudoestado de escolha onde se deve verificar se o elemento a excluir trata-se de uma pizza ou de uma bebida.

Se o elemento a excluir for uma pizza, gera-se uma transição para um estado composto contendo dois subestados. O primeiro subestado **Excluindo pizza**, exclui o objeto da classe *Pizza* por meio do método `excPizza()` e, após ser concluído, gera uma transição para o subestado **Excluindo itens de sabores**, que exclui todos os objetos da classe *Item_Sabor* associados ao objeto da classe *Pizza* por meio do método `excItemSabor()`.

Já se o elemento se referir à uma bebida, é gerada uma transição para o estado **Excluindo bebida**, onde será executado o método `excItemBebida()`, para excluir o objeto da classe *Item_Bebida* que se refere à bebida que o cliente deseja excluir.

Diagrama de Máquina de Estados Visualizar Pedidos Anteriores

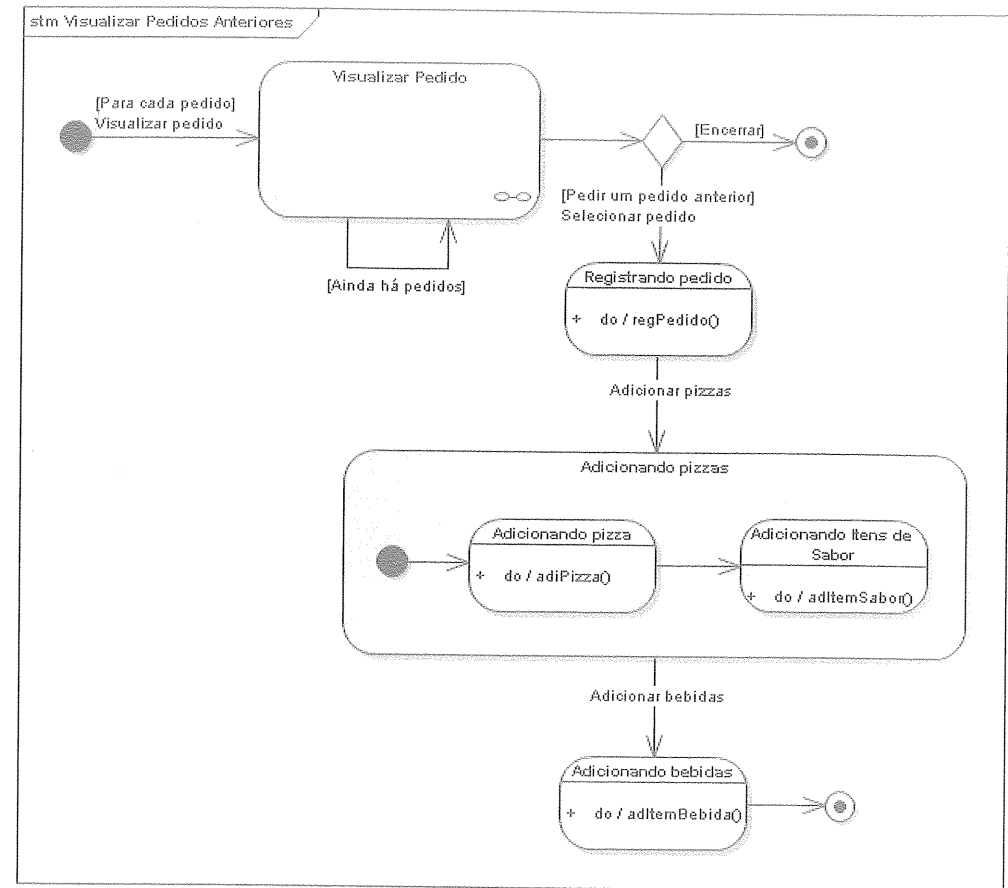


Figura 16.34 – Diagrama de Máquina de Estados Visualizar Pedidos Anteriores.

Nesse processo, é gerada uma transição para um estado de submáquina que se refere ao processo de **Visualizar Pedido**, já explicado anteriormente. Observe que existe uma autotransição para o próprio estado de submáquina, enquanto ainda houver pedidos a apresentar.

Depois que todos os pedidos anteriores tiverem sido apresentados, passa-se a um pseudoestado de escolha onde o cliente pode optar por encerrar o processo nesse momento ou solicitar um pedido já solicitado anteriormente.

Nessa segunda alternativa o cliente deve escolher um pedido, o que causa uma transição para o estado **Registrando pedido**, no qual é executado o método `regPedido()`, para instanciar um novo objeto da classe *Pedido*.

Após registrar o pedido é gerada uma transição para o estado composto **Adicionando pizzas**, que tem dois subestados. O primeiro subestado **Adicionando pizza**, executa o método `adiPizza` para instanciar um novo objeto da classe `Pizza`. Já o segundo subestado **Adicionando itens de sabor** chama o método `adItemSabor` para instanciar um novo objeto da classe `Item_Sabor` para cada sabor da pizza.

Quando o estado composto for finalizado, é gerada uma transição para o estado **Adicionando bebidas**, que dispara o método `adItemBebida` para gerar os objetos da classe `Item_Bebida` referentes ao pedido.

Diagrama de Máquina de Estados Visualizar Sabores Mais Pedidos

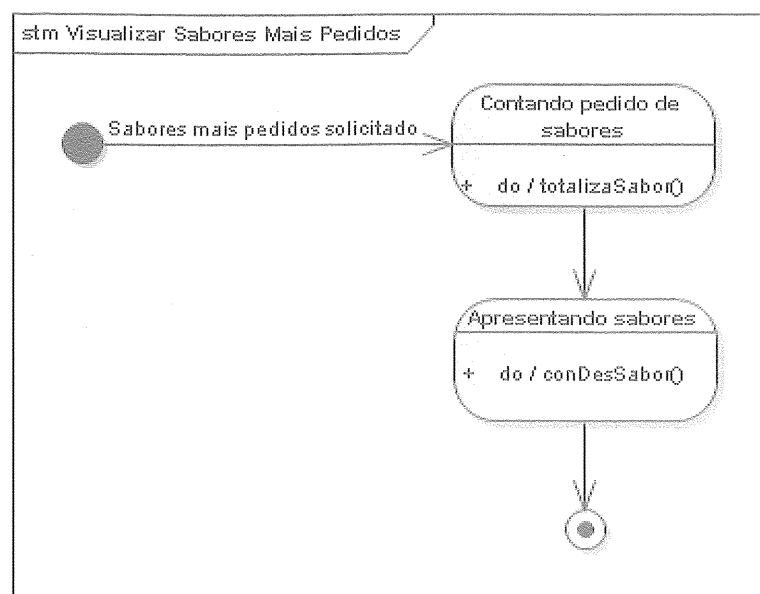


Figura 16.35 – Diagrama de Máquina de Estados Visualizar Sabores Mais Pedidos.

Quando a funcionalidade de **Visualizar Sabores Mais Pedidos** é solicitada, é gerada uma transição para o estado **Contando pedido de sabores**, que executa o método `totalizaSabor`, onde são contadas as ocorrências de todos os objetos associados a um determinado sabor.

A seguir, passa-se ao estado **Apresentando sabores**, onde é disparado o método `conDesSabor` para retornar a descrição de cada sabor a ser apresentado, junto com seus totais.

Diagrama de Máquina de Estados Concluir Pedido

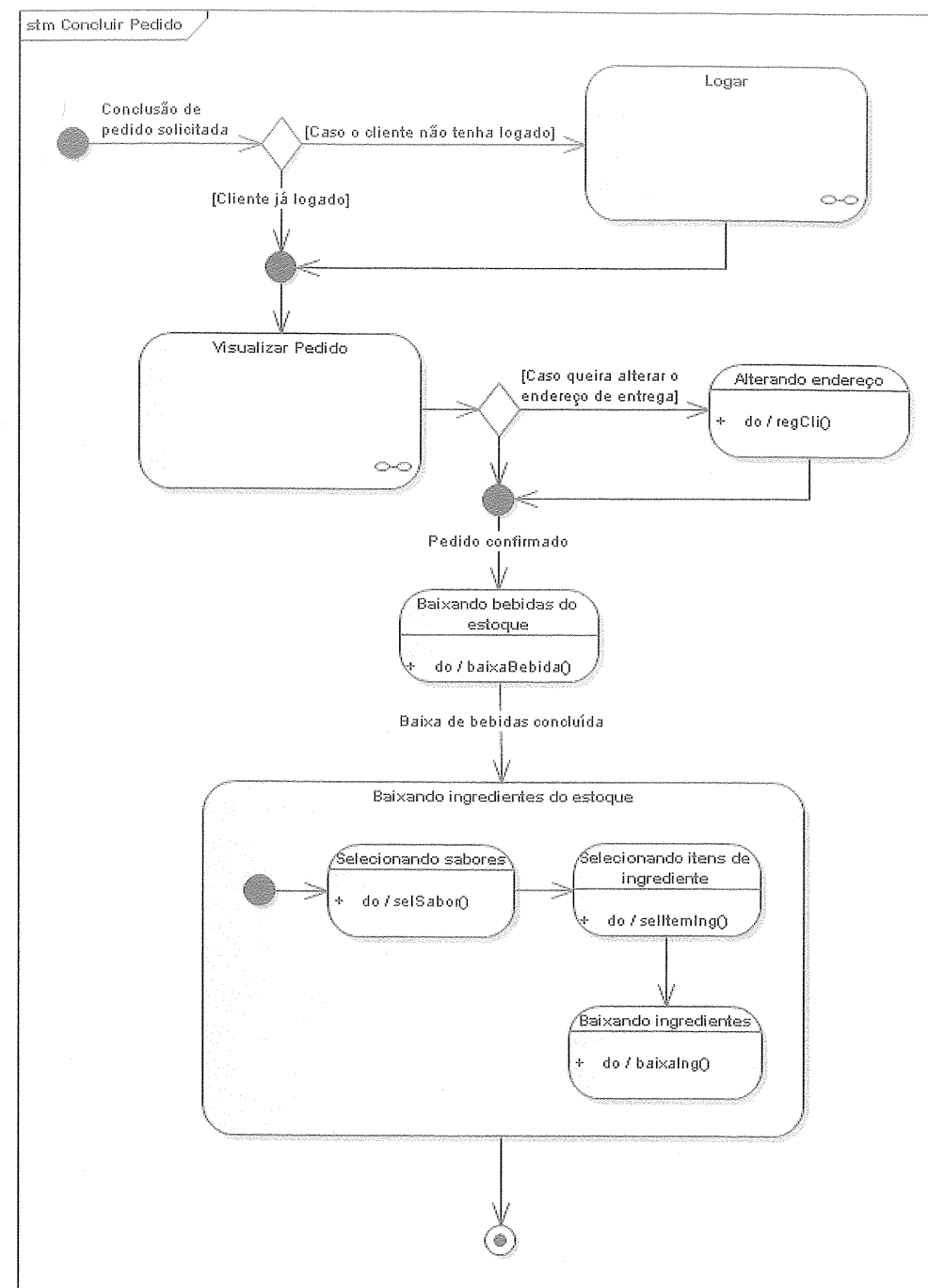


Figura 16.36 – Diagrama de Máquina de Estados Concluir Pedido.

Quando é solicitada essa funcionalidade, é gerada uma transição para um pseudoestado de escolha, que deve verificar se o cliente já está autenticado. Caso o cliente ainda não esteja logado, é gerada uma transição para um estado de submáquina referente ao processo de **Logar**, detalhado em outro diagrama.

Depois de autenticar o cliente ou caso ele já estivesse logado, passa-se a um segundo estado de submáquina que se refere ao processo de **Visualizar Pedido**, também já detalhado em outro diagrama. Isso é necessário porque se deve apresentar os detalhes do pedido mais uma vez antes de realmente concluí-lo.

A seguir, é gerada uma transição para um novo pseudoestado de escolha, que verifica se o cliente deseja alterar o endereço de entrega, caso em que é gerada uma transição para o estado **Alterando endereço**, onde é executado o método `regCli`. Observe que o fluxo que havia sido dividido pelo pseudoestado de escolha é novamente unido por um pseudoestado de junção.

A confirmação final do pedido gera uma transição para o estado **Baixando bebidas do estoque**, no qual é executado o método `baixaBebida`, que diminui a quantidade de bebidas solicitadas de seu estoque.

Ao concluir esse estado passa-se a um estado composto responsável por dar baixa nas quantidades de ingredientes necessários à produção das pizzas do pedido. Esse estado composto apresenta três subestados. No primeiro, chamado **Selecionando sabores**, é disparado o método `seleSabor` para selecionar os sabores solicitados no pedido. O segundo subestado **Selecionando itens de ingrediente**, executa o método `seleItemIng`, para selecionar os ingredientes de um sabor. Já o terceiro subestado **Baixando ingredientes**, chama o método `baixaIng`, para diminuir as quantidades necessárias à produção de cada sabor da quantidade armazenada em estoque.

Diagrama de Máquina de Estados Visualizar Pedidos em Aberto

A solicitação desse serviço produz uma transição para o estado **Consultando pedidos em aberto**, onde é disparado o método `visPedAberto`. Em seguida passa-se ao estado de submáquina **Visualizar Pedido**, que se refere ao processo de mesmo nome, já modelado em outro diagrama. Observe que esse processo é chamado para cada pedido em aberto e que há uma autotransição para o próprio estado, enquanto houver pedidos em aberto para apresentar.

Quando todos os pedidos em aberto tiverem sido visualizados, passa-se a um pseudoestado de escolha onde o funcionário deve escolher entre encerrar o processo ou finalizar um pedido. Caso o funcionário escolha a segunda alternativa, será gerada uma transição para outro estado de submáquina que se refere ao processo de **Finalizar Pedido**.

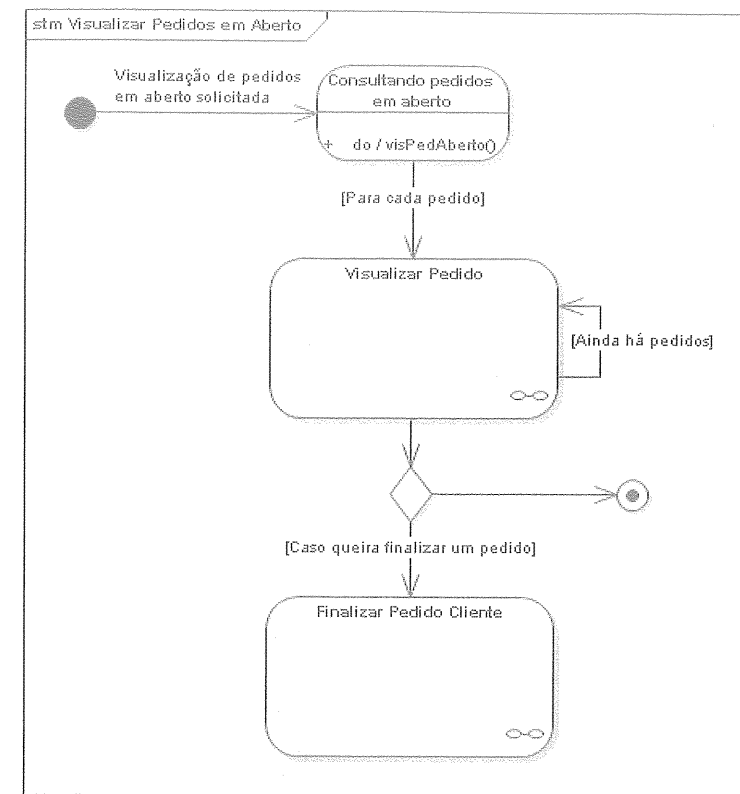


Figura 16.37 – Diagrama de Máquina de Estados Visualizar Pedidos em Aberto.

Diagrama de Máquina de Estados Finalizar Pedido Cliente

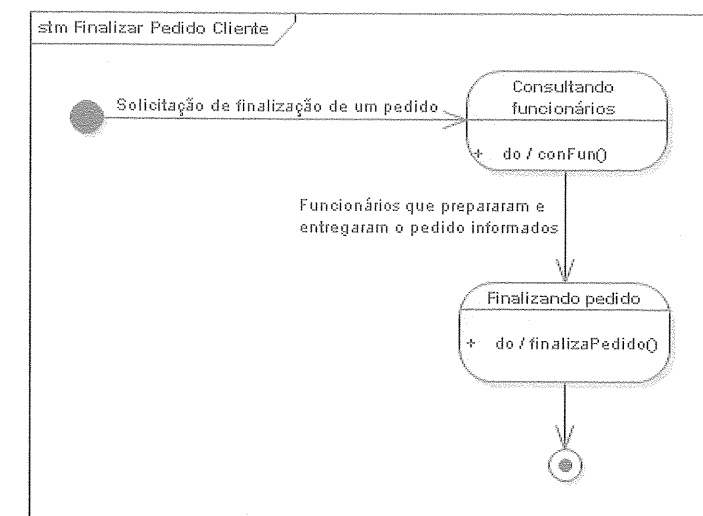


Figura 16.38 – Diagrama de Máquina de Estados Finalizar Pedido Cliente.

No momento em que esse processo é solicitado, é gerada uma transição para o estado **Consultando funcionários**, onde é executado o método `conFun` para consultar cada um dos funcionários da página e carregá-los no formulário de finalização de pedido.

A seguir, deve-se selecionar os funcionários que prepararam e entregaram o pedido, o que gerará o estado **finalizando pedido**, onde será chamado o método `FinalizaPedido`, que mudará a situação do pedido para 2, significando que esse foi atendido.

Diagrama de Máquina de Estados Manter Cardápio

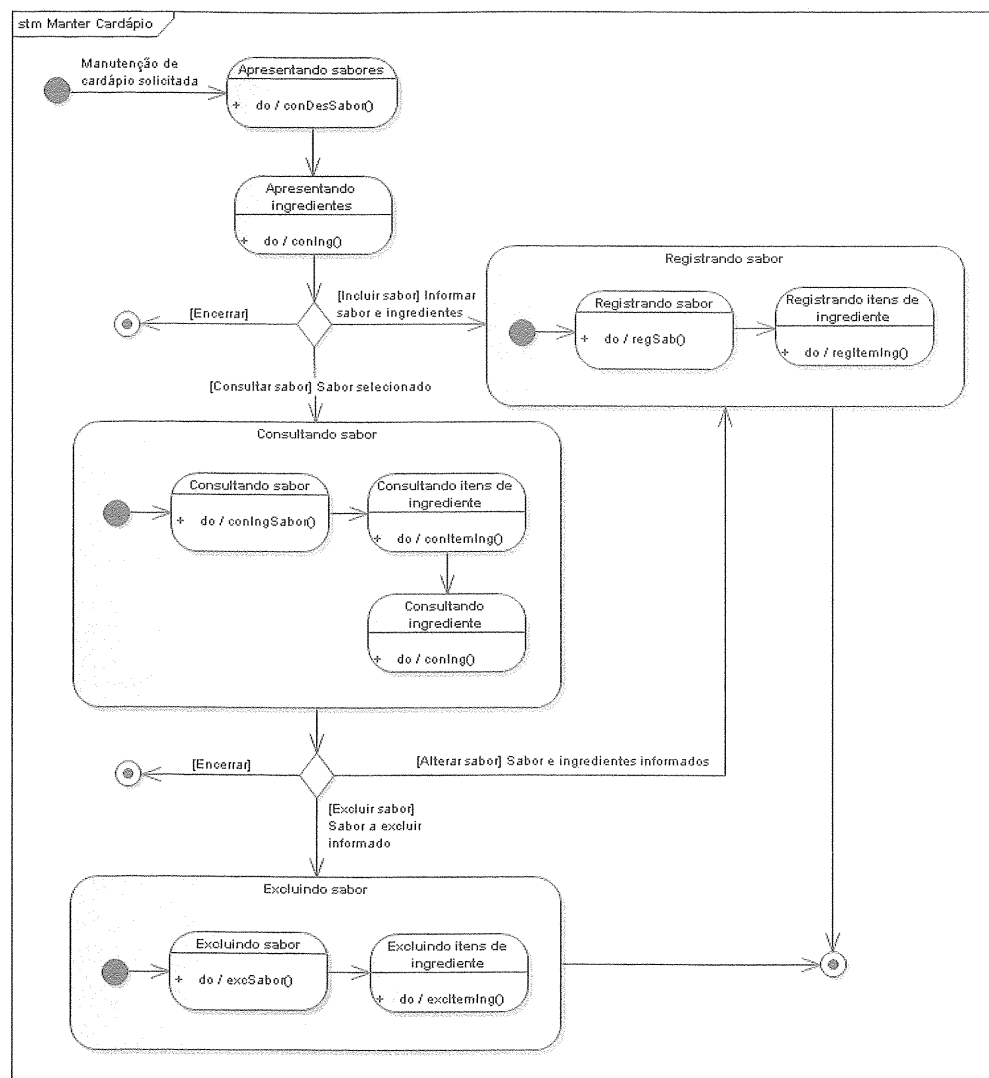


Figura 16.39 – Diagrama de Máquina de Estados Manter Cardápio.

Os estados desse processo iniciam-se com o estado **Apresentando sabores**, que executa o método `conDesSabor`, para retornar a descrição dos sabores da PizzaNet. Depois que todos os sabores tiverem sido apresentados, é gerada uma transição para o estado **Apresentando ingredientes**, que chama o método `conIng`, para retornar os ingredientes registrados no sistema.

A seguir, gera-se uma transição para um pseudoestado de escolha que representa a decisão do administrador entre registrar um novo sabor, consultar um sabor já registrado ou encerrar o processo.

Caso o administrador queira inserir um novo sabor, é gerada uma transição para o estado composto **Registrando sabor**, no interior do qual são executados os subestados **Registrando sabor**, no qual é executado o método `regSab`; e **Registrando itens de ingrediente**, em que é executado o método `regItemIng`, para instanciar novos objetos da classe `Item_Ingrediente` relacionados ao novo sabor. Após a conclusão desse estado composto, o processo é encerrado.

Caso o administrador deseje consultar um sabor existente, esse deve selecioná-lo no formulário, o que gera uma transição para o estado composto **Consultando sabor**, que contém três subestados. No primeiro **Consultando sabor**, é disparado o método `conIngSabor` para consultar os ingredientes do sabor que está sob consulta, como a classe `Sabor` não contém todas as informações necessárias a isso, é necessário gerar uma transição para o segundo subestado, chamado **Consultando itens de ingrediente**, que chama o método `conItemIng` para recuperar as informações dos objetos da classe `Item_Ingrediente` associados ao sabor. É necessária ainda uma última transição para o terceiro subestado, **Consultando ingrediente**, que executa o método `conIng` para recuperar a descrição de cada ingrediente.

Após a conclusão dessa consulta, o processo atinge um novo pseudoestado de escolha, onde se deve decidir entre alterar um sabor, excluir um sabor ou encerrar o processo.

Caso o administrador escolha alterar um sabor, ele deve alterar as informações deste, bem como, dos ingredientes necessários ao sabor. Isso gera uma transição para um comportamento idêntico ao representado no estado composto **Registrando sabor**, por esse motivo uma transição o atinge novamente.

Porém, se o administrador quiser excluir um sabor, ele deve selecioná-lo e solicitar sua exclusão no formulário, isso faz com que uma transição atinja o estado composto **Excluindo sabor**, que contém o subestado **Excluindo sabor**, no qual é executado o método `excSabor` para excluir o objeto da classe `Sabor` selecionado; e o subestado **Excluindo itens de ingrediente**, que executa o método `excItemIng`, o qual destrói todos os objetos da classe `Item_Ingrediente` associados ao objeto da classe `Sabor` em questão. Isso é necessário porque há uma associação de composição

entre a classe Sabor e a classe Item_Ingrediente, assim, quando um objeto da classe Sabor for excluído, todos os seus objetos-parte devem ser igualmente excluídos.

Diagrama de Máquina de Estados Emitir Produtos em Baixa no Estoque

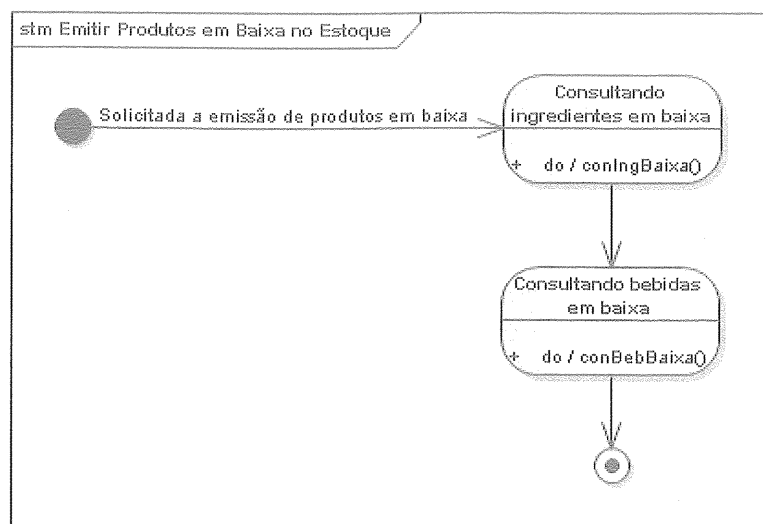


Figura 16.40 – Diagrama de Máquina de Estados Emitir Produtos em Baixa no Estoque.

Esse processo tem apenas dois estados, onde o primeiro, denominado **Consultando ingredientes em baixa**, dispara o método `conIngBaixa`, para retornar todos os ingredientes cuja quantidade em estoque esteja abaixo da mínima.

Depois que o estado estiver concluído, passa-se ao estado **Consultando bebidas em baixa**, no qual é chamado o método `conBebBaixa`, para retornar as bebidas cuja quantidade em estoque estão na mesma situação.

Diagrama de Máquina de Estados Emitir Compras em Aberto

O disparo desse processo gera uma transição para o estado **Consultando compras em aberto**, que irá executar o método `comprasAbertas`. Em seguida, passa-se ao estado **Consultando fornecedor**, que executa o método `conFor` para retornar os dados do fornecedor a que cada compra se refere.

A seguir, passa-se ao estado composto **Consultando ingredientes pedidos**, cujo primeiro subestado **Selecionando itens de ingrediente da compra** executa o método `sel_ItemCompraIng` para retornar todos os objetos da classe `Item_Compra_Ing` associados ao objeto da classe `Compra`. Já o segundo subestado **Consultando ingrediente**, chama o método `conIng` para recuperar a descrição de cada ingrediente.

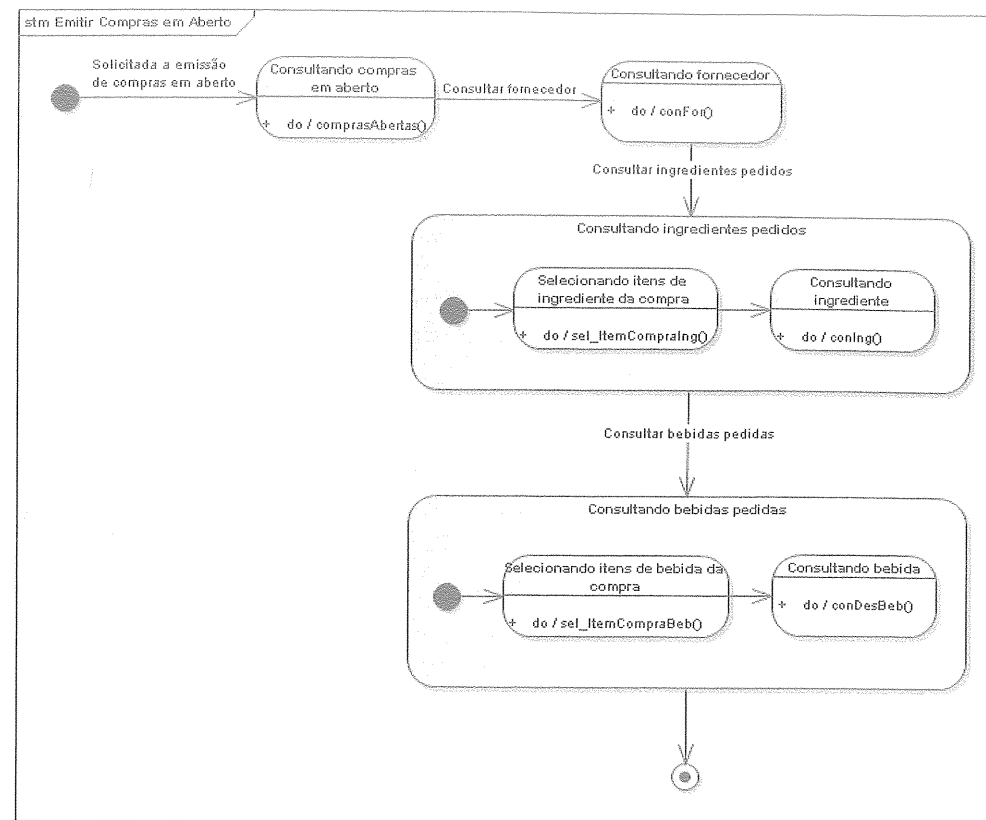


Figura 16.41 – Diagrama de Máquina de Estados Emitir Compras em Aberto.

O estado composto seguinte, chamado **Consultando bebidas pedidas** é semelhante ao anterior, contendo o subestado **Selecionando itens de bebida da compra**, que dispara o método `sel_ItemCompraBeb`, para selecionar todos os objetos da classe `Item_Compra_Beb` associados à compra; esse estado composto contém também o subestado **Consultando bebida**, que chama o método `conDesBeb`, cuja função é retornar a descrição da bebida relacionada ao objeto da classe `Item_Compra_Beb`.

Diagrama de Máquina de Estados Manter Compras Fornecedor

Esse processo se inicia com uma transição para um estado de submáquina que se refere ao processo **Emitir Compras em Aberto**, explicado na subseção anterior.

A partir da listagem apresentada por esse estado de submáquina, o administrador deve tomar uma decisão, aqui representada por um pseudoestado de escolha, onde ele deve optar por finalizar uma compra, montar um novo pedido de compra ou encerrar o processo.

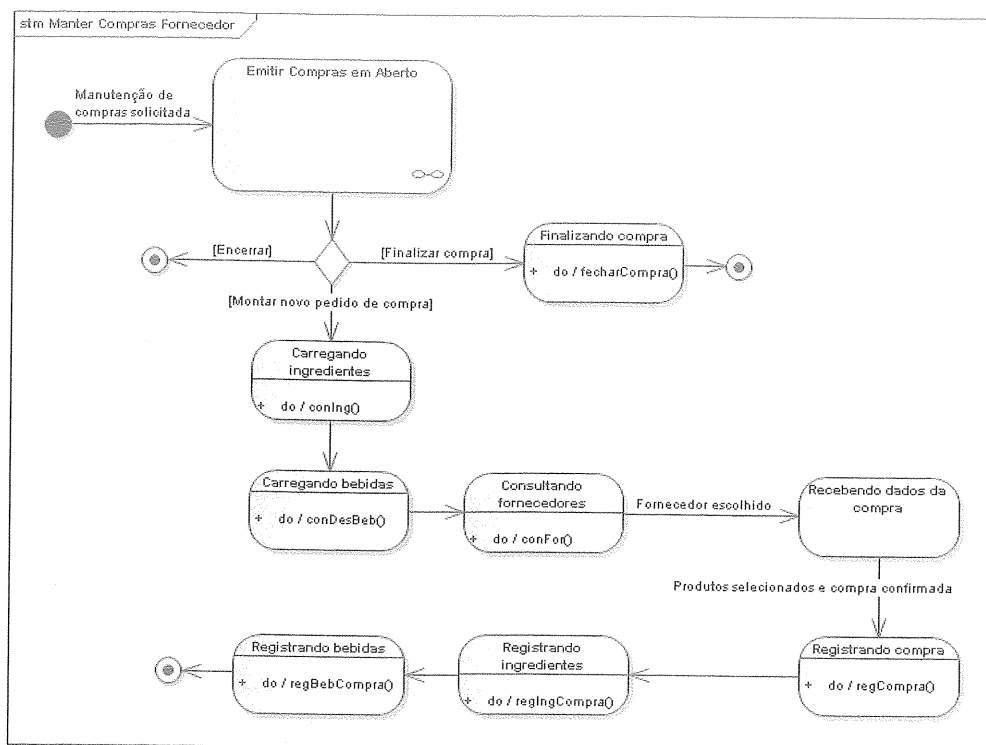


Figura 16.42 – Diagrama de Máquina de Estados Manter Compras Fornecedor.

Se o administrador desejar finalizar uma compra, ele deve selecionar a compra em questão e marcá-la como finalizada, o que gerará uma transição para o estado **Finalizando compra**, no qual será executado o método `fecharCompra`, que mudará a situação da compra para 1, significando que ela se encontra finalizada.

Porém, se o administrador quiser montar um novo pedido de compra, primeiramente será gerada uma transição para o estado **Carregando ingredientes**, onde será executado o método `conIng` para recuperar a descrição de todos os ingredientes utilizados pela pizzeria.

A seguir, passa-se a um estado semelhante, chamado **Carregando bebidas**, no qual é executado o método `conDesBeb` para retornar a descrição de todas as bebidas comercializadas pela PizzaNet.

A seguir, gera-se um novo estado, denominado **Consultando fornecedores**, em que é chamado o método `conFor` para retornar os dados de todos os fornecedores que trabalham com a empresa.

A partir daí passa-se ao estado **Recebendo dados da compra**, no qual o estado ficará aguardando que o administrador informe os dados da compra, isso é, selecione os possíveis ingredientes e bebidas desejados, bem como selecione o fornecedor para o qual solicitará esses itens.

Quando isso ocorre, é gerada uma transição para o estado **Registrando compra**, em que será disparado o método `regCompra`, para instanciar um novo objeto da classe compra e, em seguida, será gerada uma transição para o estado **Registrando ingredientes**, no qual é chamado o método `regIngCompra`, para instanciar os objetos da classe `Item_Compra_Ing`, passando a seguir para o estado **Registrando bebidas**, que executa o método `regBebCompra`, que faz o mesmo com relação aos objetos da classe `Item_Compra_Beb`.

Diagrama de Máquina de Estados Emitir Melhores Clientes

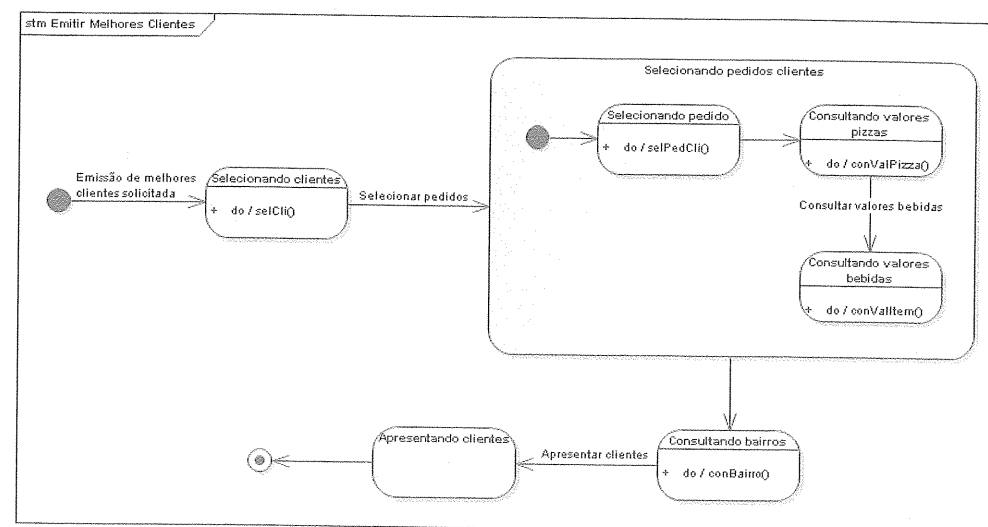


Figura 16.43 – Diagrama de Máquina de Estados Emitir Melhores Clientes.

O primeiro estado desse processo, denominado **Selecionando clientes**, executa o método `selCli`, para selecionar todos os clientes registrados na empresa.

Após a finalização deste estado, passa-se ao estado composto **Selecionando pedidos clientes**, o qual contém três subestados, onde no primeiro, denominado **Selecionando pedido** será disparado o método `selPedCli`, para selecionar todos os pedidos de um dos clientes selecionados e somar seus valores. Como os objetos da classe `Pedido` não possuem todas as informações necessárias a esse processo, é necessário disparar uma transição para o estado **Consultando valores pizzas**, que chama o método `conValPizza`, para retornar o valor de cada pizza solicitada no pedido. Quando esse estado se conclui, é gerada uma transição para o estado **Consultando valores bebidas**, onde é executado o método `conValItem`, para retornar o valor de cada bebida solicitada no pedido.

Após este estado composto ser concluído, passa-se ao estado **Consultando bairros**, e executa-se o método `conBairro` para cada cliente totalizado. Finalmente

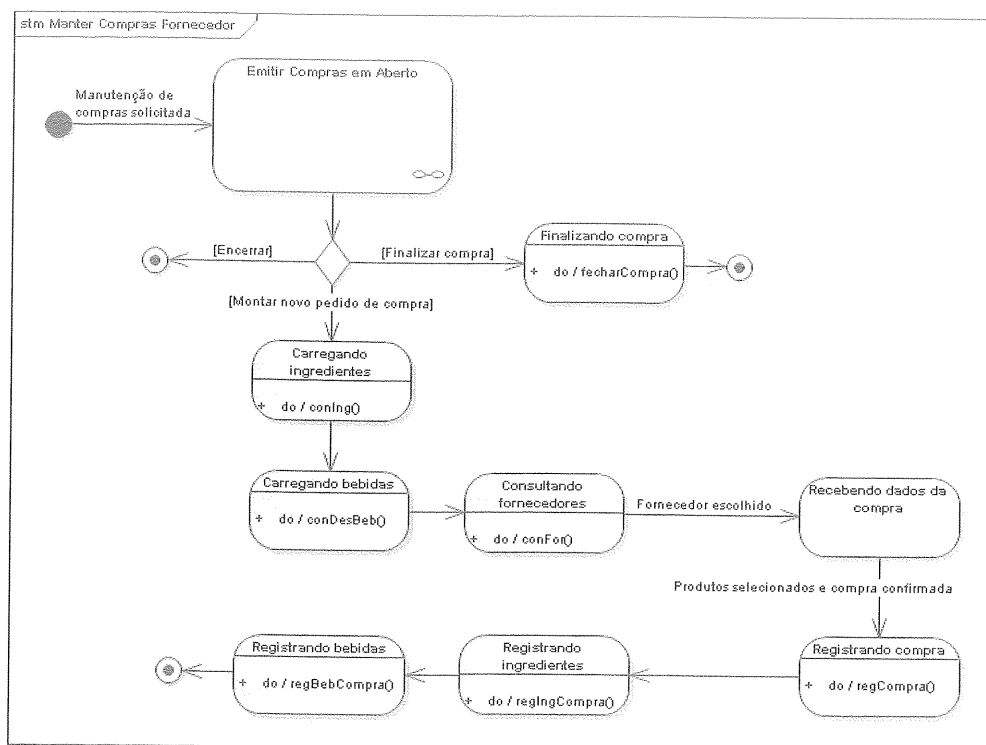


Figura 1642 – Diagrama de Máquina de Estados Manter Compras Fornecedor.

Se o administrador desejar finalizar uma compra, ele deve selecionar a compra em questão e marcá-la como finalizada, o que gerará uma transição para o estado **Finalizando compra**, no qual será executado o método `fecharCompra`, que mudará a situação da compra para 1, significando que ela se encontra finalizada.

Porém, se o administrador quiser montar um novo pedido de compra, primeiramente será gerada uma transição para o estado **Carregando ingredientes**, onde será executado o método `conIng` para recuperar a descrição de todos os ingredientes utilizados pela pizzeria.

A seguir, passa-se a um estado semelhante, chamado **Carregando bebidas**, no qual é executado o método `conDesBeb` para retornar a descrição de todas as bebidas comercializadas pela PizzaNet.

A seguir, gera-se um novo estado, denominado **Consultando fornecedores**, em que é chamado o método `conFor` para retornar os dados de todos os fornecedores que trabalham com a empresa.

A partir daí passa-se ao estado **Recebendo dados da compra**, no qual o estado ficará aguardando que o administrador informe os dados da compra, isso é, selecione os possíveis ingredientes e bebidas desejados, bem como selecione o fornecedor para o qual solicitará esses itens.

Quando isso ocorre, é gerada uma transição para o estado **Registrando compra**, em que será disparado o método `regCompra`, para instanciar um novo objeto da classe compra e, em seguida, será gerada uma transição para o estado **Registrando ingredientes**, no qual é chamado o método `regIngCompra`, para instanciar os objetos da classe `Item_Compra_Ing`, passando a seguir para o estado **Registrando bebidas**, que executa o método `regBebCompra`, que faz o mesmo com relação aos objetos da classe `Item_Compra_Beb`.

Diagrama de Máquina de Estados Emitir Melhores Clientes

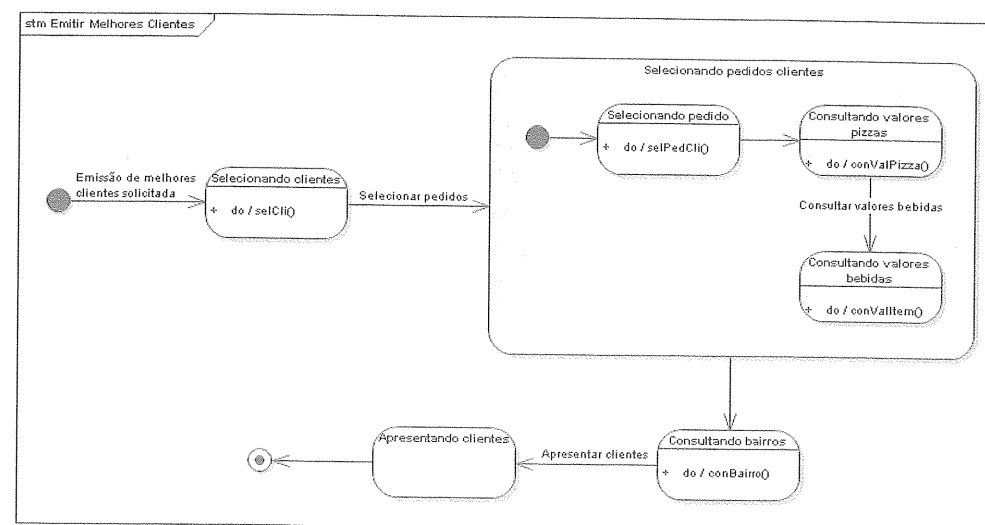


Figura 1643 – Diagrama de Máquina de Estados Emitir Melhores Clientes.

O primeiro estado desse processo, denominado **Selecionando clientes**, executa o método `selCli`, para selecionar todos os clientes registrados na empresa.

Após a finalização deste estado, passa-se ao estado composto **Selecionando pedidos clientes**, o qual contém três subestados, onde no primeiro, denominado **Selecionando pedido** será disparado o método `selPedCli`, para selecionar todos os pedidos de um dos clientes selecionados e somar seus valores. Como os objetos da classe `Pedido` não possuem todas as informações necessárias a esse processo, é necessário disparar uma transição para o estado **Consultando valores pizzas**, que chama o método `conValPizza`, para retornar o valor de cada pizza solicitada no pedido. Quando esse estado se conclui, é gerada uma transição para o estado **Consultando valores bebidas**, onde é executado o método `conValItem`, para retornar o valor de cada bebida solicitada no pedido.

Após este estado composto ser concluído, passa-se ao estado **Consultando bairros**, e executa-se o método `conBairro` para cada cliente totalizado. Finalmente

passa-se ao estado **Apresentando clientes**, onde os clientes serão apresentados na interface por ordem de maior consumo.

Diagrama de Máquina de Estados Emitir Consumo por Período

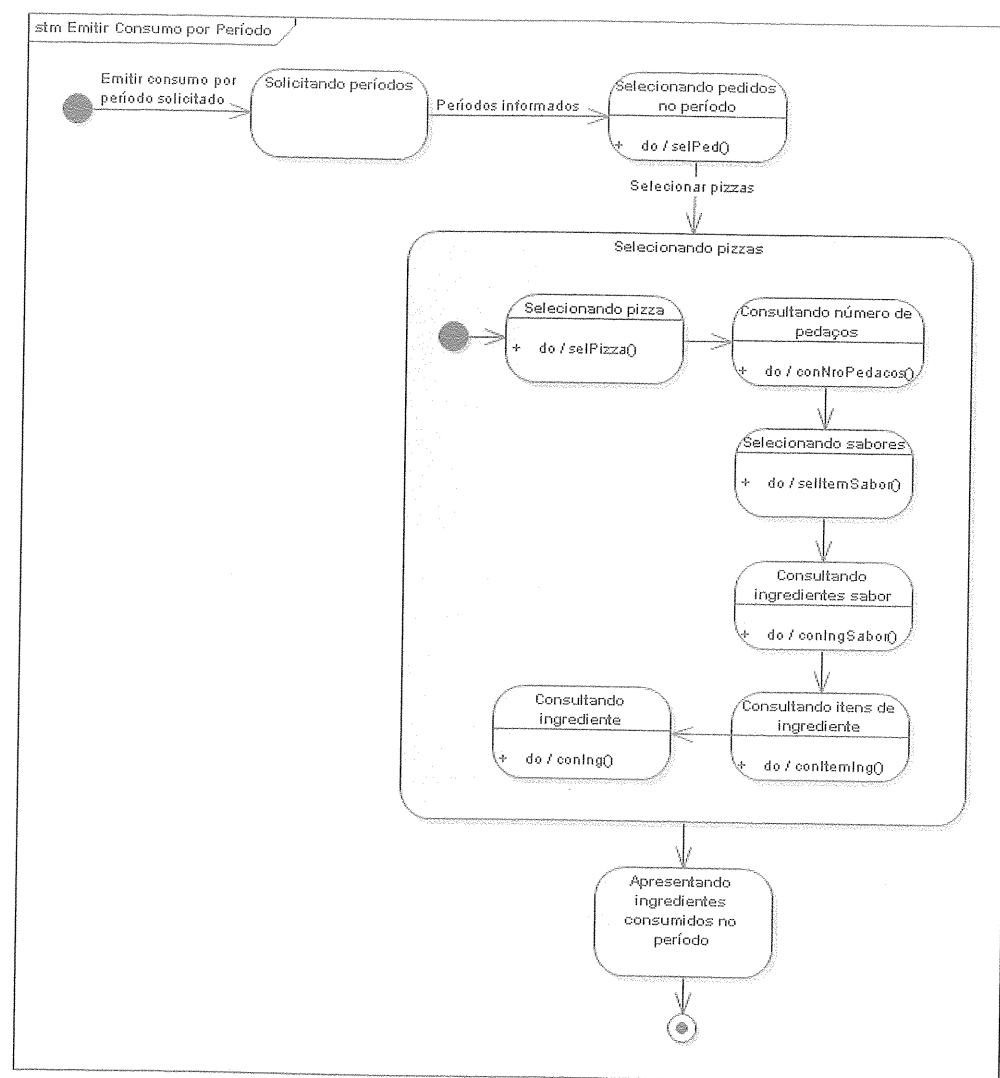


Figura 16.44 – Diagrama de Máquina de Estados Emitir Consumo por Período.

Os estados desse processo se iniciam quando o administrador solicita a listagem dos melhores clientes. Quando isso ocorre, é gerada uma transição para o estado **Solicitando períodos**, que ficará aguardando que o administrador informe as datas desejadas. Quando estas forem fornecidas é gerada uma transição para o estado **Selecionando pedidos no período**, no qual é executado o método `selPed` para retornar todos os pedidos no período.

A seguir, passa-se ao estado composto **Selecionando pizzas**, no qual, em seu primeiro subestado denominado **Selecionando pizza**, é disparado o método `selPizza`, para selecionar as pizzas de um pedido específico. O subestado seguinte **Consultando número de pedaços**, chama o método `conNroPedacos`, para retornar o número de pedaços da pizza. Em seguida, passa-se ao subestado **Selecionando sabores**, no qual é executado o método `selItemSabor`, para selecionar os objetos da classe `Item_Sabor` associados ao objeto da classe `Pizza`; passando-se para o estado **Consultando ingredientes sabor**, que dispara o método `conIngSabor`, para retornar os ingredientes associados ao sabor. A seguir, chega-se ao estado **Consultando itens de ingrediente**, onde é chamado o método `conItemIng`, que retorna todos os objetos da classe `Item_Ingrediente` associados ao objeto `Sabor`. Finalmente, atinge-se o último subestado, chamado **Consultando ingrediente**, que dispara o método `conIng`, para retornar a descrição de cada ingrediente.

Quando esse estado composto for concluído, gera-se uma transição para o estado **Apresentando ingredientes consumidos no período**, onde são apresentados os totais dos ingredientes consultados e encerra-se o processo.

16.2.9 Diagramas de Atividade da PizzaNet

Nesta seção serão modelados os algoritmos dos processos já apresentados anteriormente, por meio do diagrama de atividade. O leitor notará que muitos dos diagramas apresentados nesta seção estão bastante detalhados, pois o diagrama de atividade é um dos diagramas de maior detalhamento da UML. Em alguns casos, os diagramas poderiam até mesmo ser simplificados. No entanto, as instruções representadas pelos diagramas desta seção são gerais, uma vez que estão dissociadas de uma linguagem de programação específica.

Diagrama de Atividade Escolher Pizza

A primeira ação modelada nesse diagrama representa a seleção de todos os sabores registrados no sistema. Essas informações são recuperadas a partir de objetos da classe `Sabor`, como demonstra a direção do fluxo de objetos. Na próxima ação a atividade posiciona-se sobre o primeiro objeto da classe `Sabor` encontrado e a ação seguinte retorna a descrição do sabor. Na verdade essa última ação, a rigor, também deveria ter um fluxo de objetos com um objeto da classe `Sabor`, mas optamos por apresentar somente as operações mais importantes nesses diagramas.

A seguir, a atividade encontra um nó de decisão, que deve verificar se ainda existem sabores a apresentar. Se isso for verdadeiro, a atividade posiciona-se sobre o próximo sabor e volta a retornar sua descrição, ficando nesse laço enquanto houver sabores a apresentar.

Quando o laço se encerrar, passa-se à ação em que a atividade ficará aguardando que o usuário escolha o tamanho da pizza que deseja. Quando este for escolhido, a atividade passa à ação que consulta a quantidade de sabores permitidos para o tamanho. A seguir, passa-se à ação na qual a atividade recebe os sabores desejados pelo cliente.

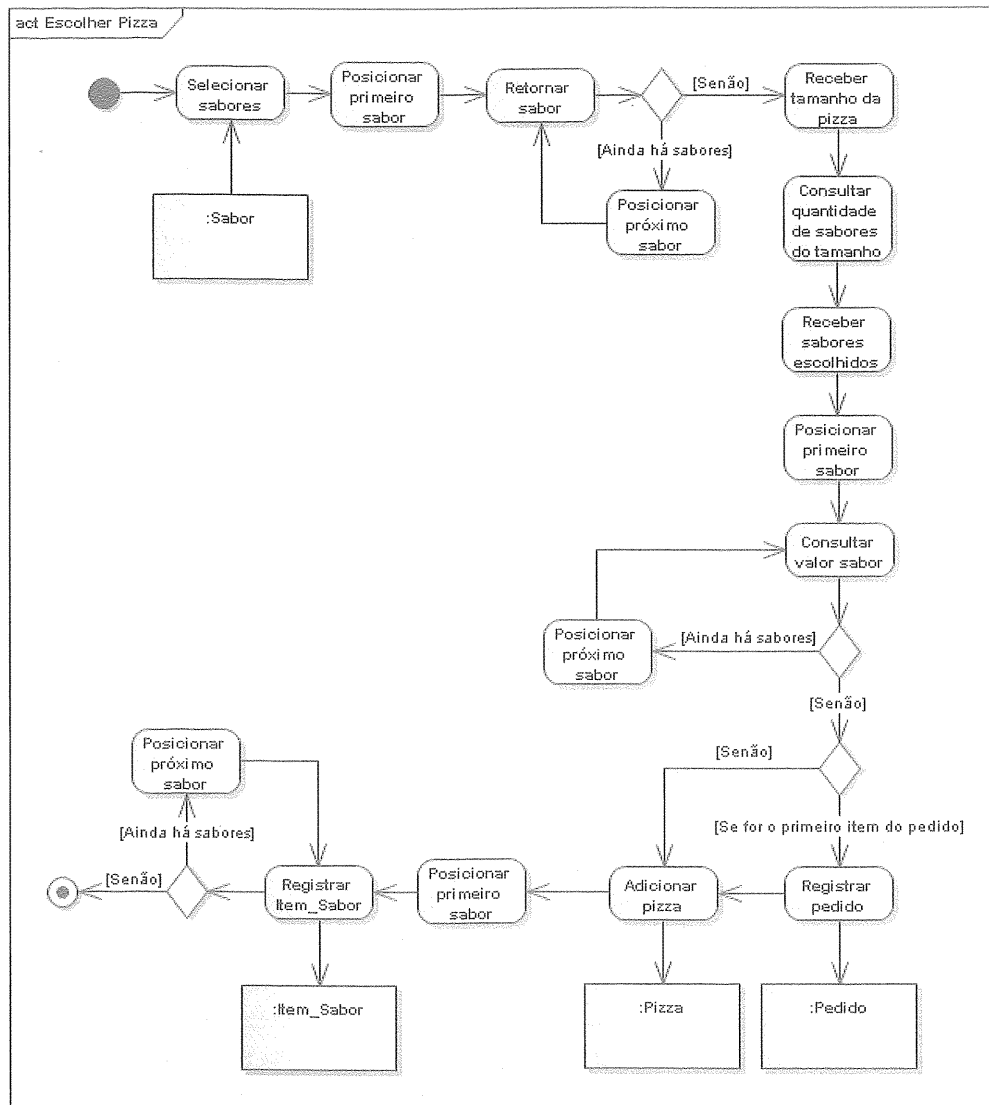


Figura 1645 – Diagrama de Atividade Escolher Pizza.

No momento em que os sabores são escolhidos, a atividade executa a ação em que posiciona-se sobre o primeiro sabor escolhido e, em seguida, consulta seu valor, passando para um nó de decisão que determina se ainda há sabores a consultar, caso em que a atividade se posiciona sobre o próximo sabor e repete a operação.

Caso não haja mais sabores, a atividade atinge outro nó de decisão, que deve verificar se a pizza escolhida é o primeiro item do pedido. Se isso for verdadeiro, é executada a ação **Registrar pedido**, que instancia um novo objeto da classe **Pedido**, conforme demonstra o fluxo de objetos.

A seguir, ou se a pizza não se constituir no primeiro item do pedido, a atividade passa à ação em que a pizza é adicionada ao pedido, gerando-se um novo objeto da classe **Pizza**. Em seguida a atividade se posiciona sobre o primeiro sabor escolhido e inicia um laço onde o sabor é registrado, gerando-se um novo objeto da classe **Item_Sabor**, passando-se a um nó de decisão que verifica se ainda há sabores a registrar. Em caso positivo, a atividade se posiciona sobre o próximo sabor, registrando-o a seguir. Esse laço é executado enquanto houver sabores, no momento em que não houver mais sabores a atividade será encerrada.

Diagrama de Atividade Escolher Bebida

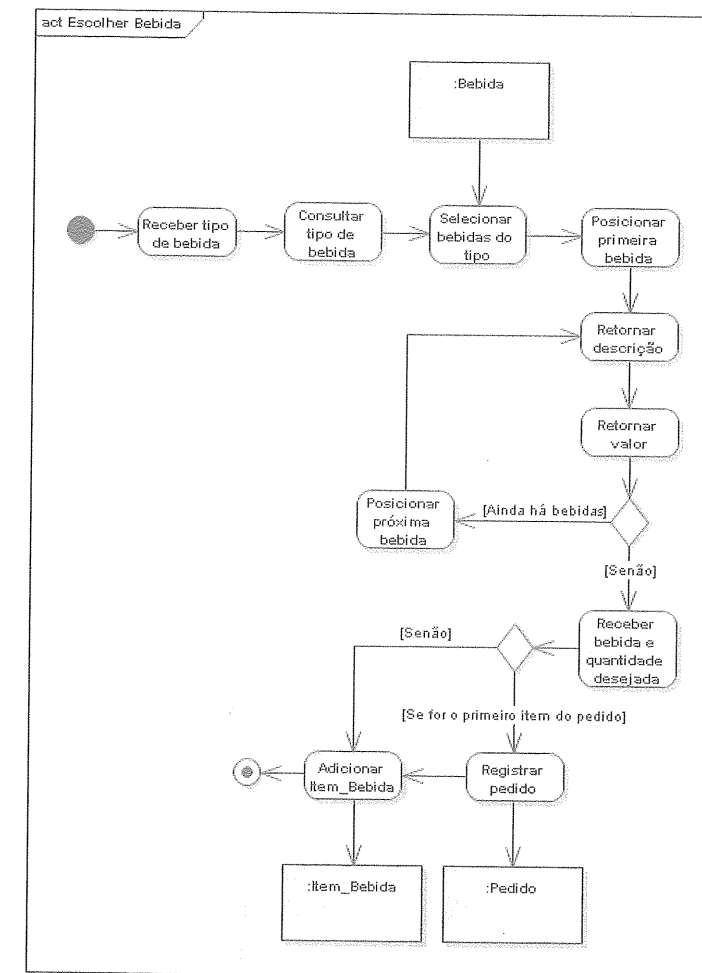


Figura 1646 – Diagrama de Atividade Escolher Bebida.

A primeira ação modelada nesse diagrama refere-se ao recebimento do tipo de bebida escolhido pelo cliente. A ação seguinte consulta o tipo de bebida escolhido e passa a ação em que serão selecionadas todas as bebidas do tipo escolhido. Observe que existe um fluxo de objetos nessa ação e que a seta desse fluxo indica que as informações são transmitidas de objetos da classe *Bebida* para a ação. A rigor, a ação anterior também deveria ter um fluxo de objetos com um objeto da classe *Tipo_Bebida*, mas preferimos identificar somente os fluxos mais importantes nesses diagramas.

A seguir, a atividade posiciona-se sobre o primeiro objeto da classe *Bebida* encontrado, passando para a ação em que é retornada a descrição da bebida e, em seguida, à ação em que é retornado o valor da bebida em questão. A seguir, a atividade atinge um nó de decisão, onde se deve determinar se ainda há bebidas a pesquisar, caso em que a atividade posiciona-se sobre a próxima bebida e repete a operação, ficando nesse laço enquanto houver bebidas a consultar.

Quando não houver mais bebidas a apresentar, a atividade passa a ação em que será recebida a bebida e a quantidade escolhida pelo cliente, passando-se para um nó de decisão que verifica se a bebida selecionada constitui-se no primeiro item do pedido. Nesse caso, a atividade executa a ação **Registrar pedido**, onde é instanciado um novo objeto da classe *Pedido*, como demonstra o fluxo de objetos.

Depois disso, ou se a bebida não for o primeiro item do pedido, a atividade passa à ação em que é instanciado um novo objeto da classe *Item_Bebida*, referente a bebida selecionada.

Diagrama de Atividade Logar

Esse diagrama contém duas atividades, uma apenas referenciada, sem detalhar seus nós de ação, e outra detalhando suas ações internas.

O diagrama inicia-se com um nó de decisão, cuja função é determinar se o cliente já se encontra registrado. Se o cliente não estiver registrado, o fluxo atinge a atividade **Autorregistrar**, cujas ações não estão detalhadas nesse diagrama.

Caso o cliente já se encontre cadastrado, é executada a atividade **Logar**. Nessa atividade, primeiramente são recebidos o nome-*login* e a senha. A seguir, passa-se à ação **validar login**, para determinar se esse é um nome-*login* válido, o que leva a um nó de decisão que deve verificar se o nome-*login* foi considerado válido.

Se o nome-*login* for considerado inválido, a atividade é encerrada, caso contrário passa-se à ação em que a senha é validada, o que leva a um segundo nó de decisão, onde é verificado se a senha fornecida é uma senha válida. Caso a senha não seja válida, a atividade é encerrada, caso contrário executa-se a ação **Logar**, que autentica o cliente no sistema e encerra-se a atividade.

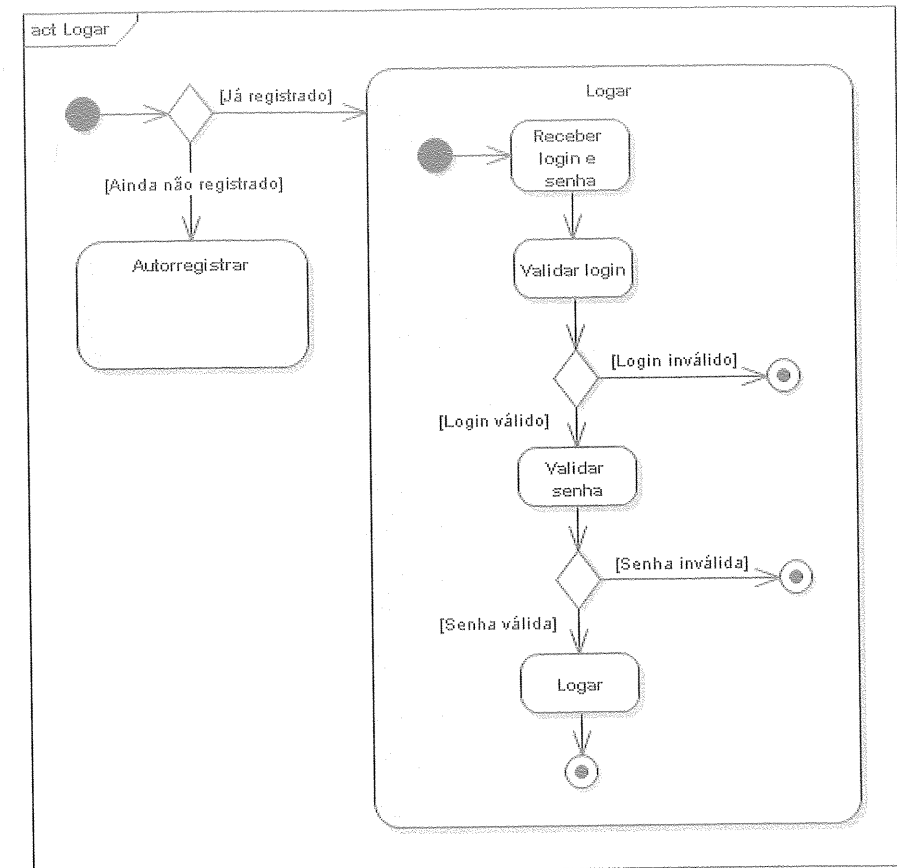


Figura 1647 – Diagrama de Atividade Logar.

Diagrama de Atividade Autorregistrar

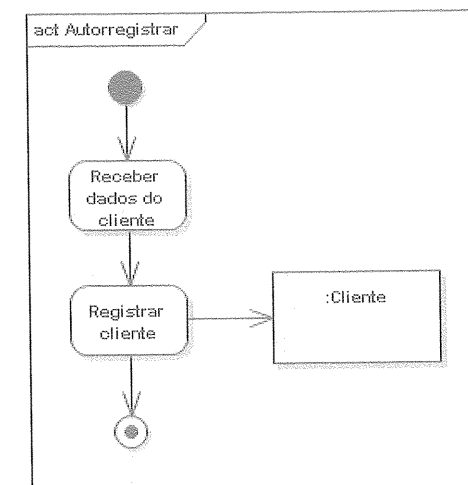


Figura 1648 – Diagrama de Atividade Autorregistrar.

Esta é uma atividade muito simples, composta de apenas dois nós de ação. No primeiro são recebidos os dados do cliente e no segundo este é registrado, gerando-se uma nova instância da classe *Cliente*, conforme demonstra o fluxo de objetos entre essa ação e o objeto da classe *Cliente*.

Diagrama de Atividade Opinar

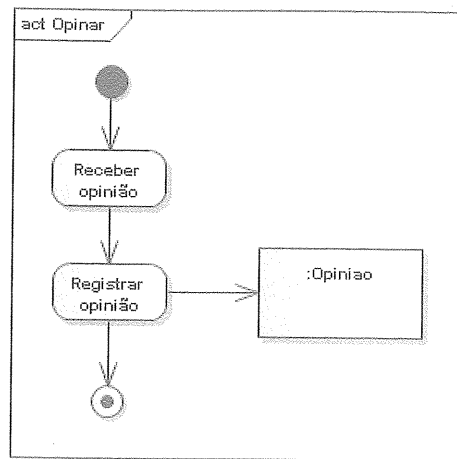


Figura 16.49 – Diagrama de Atividade Opinar.

Essa atividade é bastante semelhante à anterior, composta igualmente de dois nós de ação. No primeiro nó de ação, é recebida a opinião do cliente e no segundo esta é registrada, gerando um novo objeto da classe *Opiniao*.

Diagrama de Atividade Visualizar Pedido

Esse diagrama contém duas atividades, sendo que a segunda é apenas referenciada, sem detalhar seus nós de ação. A atividade principal, denominada **Visualizar Pedido**, inicia-se pelo nó de ação que consulta o pedido a ser visualizado. O leitor perceberá que existe um fluxo de objeto entre essa ação e um objeto da classe *Pedido*, sobre o qual é realizada essa operação.

Aqui cabe um comentário, temos evitado nos diagramas de atividade, inserir muitos fluxos de objeto para não deixá-los carregados demais, porém, como essa atividade envolve muitos objetos de diferentes classes, preferimos nesse caso, detalhar todos os fluxos de objeto desse diagrama, o que também permite ilustrar um desses casos ao leitor. Em outros diagramas, no entanto, tentaremos modelar somente os fluxos de objeto mais importantes, detalhando todos os fluxos de objeto, somente quando isso for realmente necessário para a compreensão do diagrama, como nesse caso.

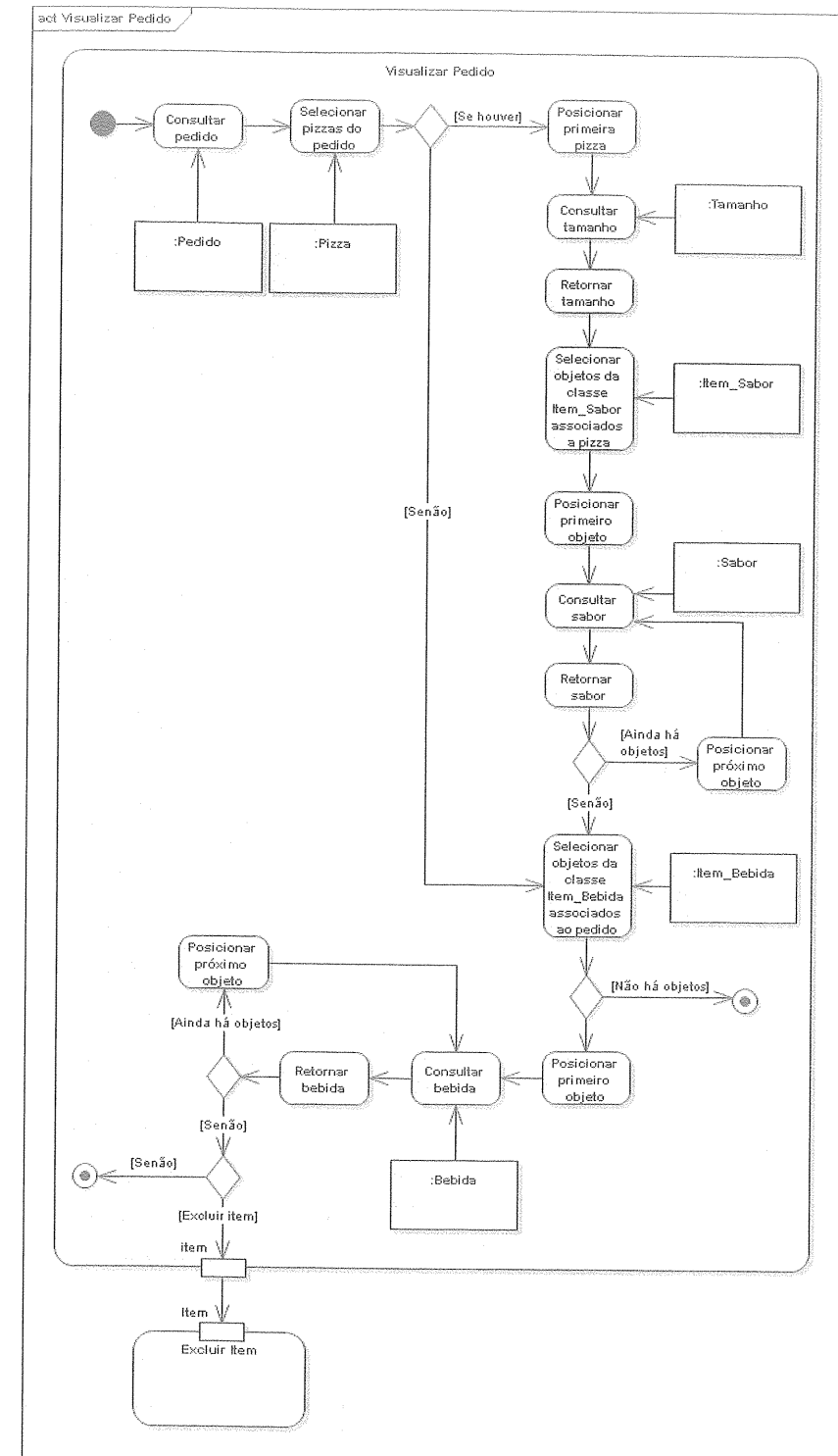


Figura 16.50 – Diagrama de Atividade Visualizar Pedido.

Após a consulta do pedido, passa-se a ação que seleciona as pizzas do pedido consultado. Note que essa informação é recuperada por meio de um fluxo de objetos, a partir de objetos da classe Pizza. Logo após isso, a atividade encontra um nó de decisão, onde é verificado se o pedido inclui pizzas. Em caso afirmativo, é executada a ação em que a atividade posiciona-se sobre o primeiro objeto da classe Pizza. Na ação seguinte, é consultado o tamanho da pizza, demonstrando por um fluxo de objetos, e passa-se à ação onde esse tamanho é retornado.

A seguir, são selecionados todos os objetos da classe Item_Sabor associados ao objeto da classe Pizza. Observe que há um fluxo de objetos entre essa ação e um objeto da classe Item_Sabor para representar essa seleção. Em seguida, a atividade posiciona-se sobre o primeiro objeto da classe Item_Sabor e, na ação seguinte, é consultada a descrição desse Sabor, buscada a partir de um objeto da classe Sabor, como demonstra o fluxo de objetos. A ação seguinte retorna a descrição do sabor consultado.

Em seguida a atividade encontra um nó de decisão que verifica se ainda há objetos da classe Item_Sabor para apresentar. Em caso positivo, a atividade posiciona-se sobre o próximo objeto e repete a operação descrita anteriormente, ficando nesse laço enquanto houver objetos a apresentar.

Quando não houver mais objetos da classe Item_Sabor a apresentar (ou se não existirem pizzas no pedido), é executada a ação Selecionar objetos da classe Item_Bebida associados ao pedido, que, por meio de um fluxo de objetos, seleciona todos os objetos da classe Item_Bebida associados ao objeto Pedido, consultado no início da atividade.

Se não forem encontrados objetos dessa classe associados ao pedido, o processo é encerrado. Caso contrário, a atividade posiciona-se sobre o primeiro objeto da classe Item_Bebida e, na próxima ação, consulta a descrição desse item por meio de um fluxo de objetos com um objeto da classe Bebida. A descrição da bebida é retornada pela ação seguinte.

A atividade passa então a um nó de decisão, que verifica se ainda há objetos da classe Item_Bebida a apresentar. Se isso for verdadeiro, a atividade posiciona-se sobre o próximo objeto e volta a consultar o objeto Bebida a ele associado, repetindo essa operação enquanto houver objetos a apresentar.

Se não houver mais objetos a apresentar a atividade encontra outro nó de decisão, onde o cliente pode escolher entre terminar o processo ou excluir algum item apresentado. Nesse último caso o fluxo atinge um nó de parâmetro de atividade que receberá o item a excluir. Esse valor será passado como parâmetro por meio de um fluxo de objetos para a atividade Excluir Item, que o receberá através de outro nó de parâmetro de atividade. Essa atividade é responsável por excluir um item do pedido e será explicada na próxima subseção.

Diagrama de Atividade Excluir Item

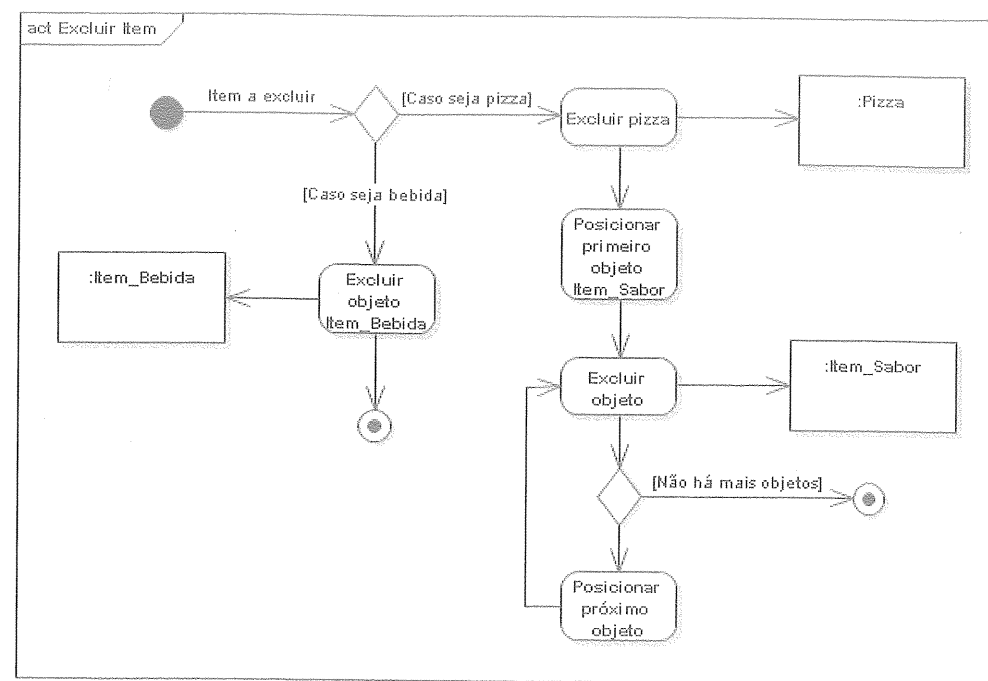


Figura 16.51 – Diagrama de Atividade Excluir Item.

Esse diagrama inicia-se com um nó de decisão onde é preciso determinar se o item a excluir trata-se de uma pizza ou de uma bebida. No primeiro caso, executa-se uma ação para excluir um objeto da classe Pizza, conforme demonstra o fluxo de objetos. Depois disso, a atividade posiciona-se sobre o primeiro objeto da classe Item_Sabor associado ao objeto da classe Pizza. Em seguida, a atividade executa uma ação para excluir o objeto da classe Item_Sabor. Observe que há um fluxo de objetos entre o nó de ação e o objeto da classe Item_Sabor, que representa a exclusão do objeto.

A seguir, a atividade encontra um nó de decisão que verifica se ainda há objetos. Se não houver mais objetos, a atividade é encerrada, caso contrário é executada uma ação que posiciona no próximo objeto e em seguida o exclui. Esse laço se repete enquanto houver objetos da classe Item_Sabor para ser excluídos.

Já se o item a excluir se tratar de uma bebida, a atividade simplesmente executa uma ação para destruir o objeto da classe Item_Bebida.

Diagrama de Atividade Visualizar Pedidos Anteriores

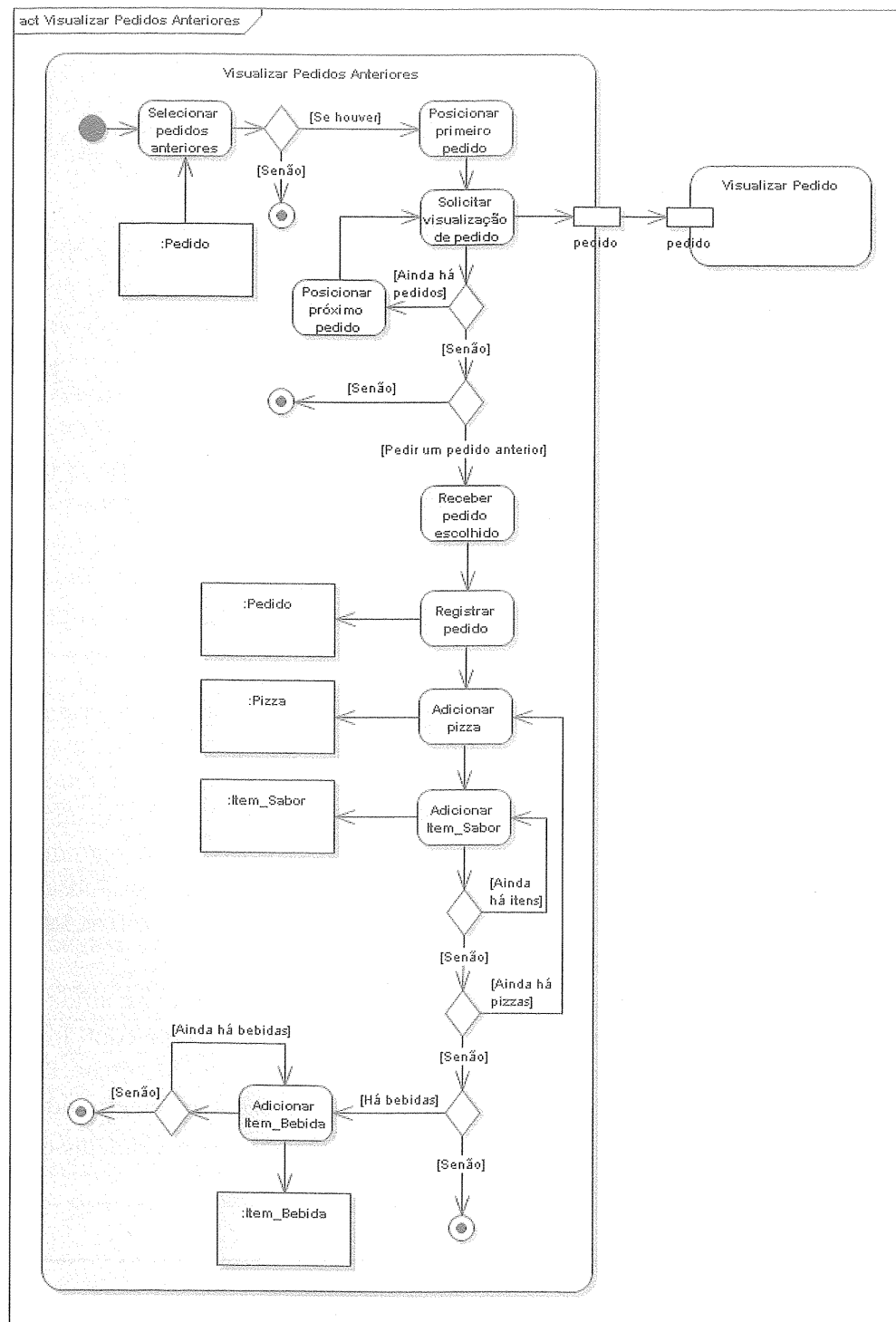


Figura 16.52 – Diagrama de Atividade Visualizar Pedidos Anteriores.

Esse diagrama é composto por duas atividades, a principal atividade, chamada *Visualizar Pedidos Anteriores* detalha seus nós de ação, enquanto a atividade *Visualizar Pedido* é apenas referenciada.

Na atividade principal, primeiramente é executada uma ação para selecionar todos os pedidos anteriores do cliente. Observe que essa pesquisa é feita por meio de um fluxo de objetos, representando a consulta a objetos da classe *Pedido*.

A seguir, a atividade passa a um nó de decisão, que deve verificar se foram encontrados pedidos anteriores do cliente. Em caso negativo, a atividade é encerrada. Caso contrário, a atividade posiciona-se sobre o primeiro objeto da classe *Pedido* encontrado e, na ação seguinte, solicita a visualização desse pedido. Como o processo de visualização de um pedido já foi modelado anteriormente em outro diagrama, passa-se o número do pedido por meio de um nó de parâmetro de atividade para a atividade *Visualizar Pedido*, que se encarregará de detalhá-lo.

A seguir, a atividade encontra um nó de decisão, que testa se ainda há pedidos a apresentar. Em caso positivo a atividade posiciona-se sobre o próximo objeto da classe *Pedido* e volta a solicitar sua visualização, permanecendo nesse laço enquanto houver pedidos a apresentar.

Quando esse laço se encerra, passa-se a um novo nó de decisão, que representa a escolha do cliente, que deve decidir entre encerrar o processo ou pedir um pedido novamente. Caso o cliente escolha a segunda alternativa, a atividade receberá o pedido escolhido pelo cliente, passando a seguir para a ação que registra o novo pedido, instanciando um novo objeto da classe *Pedido*, como demonstra o fluxo de objetos.

Depois disso, a atividade irá registrar a primeira pizza do pedido, instanciando um novo objeto da classe *Pizza*, como demonstra o fluxo de objetos e, em seguida, irá registrar o primeiro sabor da pizza, instanciando um novo objeto da classe *Item_Sabor*. A seguir, um nó de decisão verifica se ainda há sabores associados à pizza, caso em que se repete a última ação. Se não houver mais sabores, um novo nó de decisão testa se ainda há pizzas associadas ao pedido, situação em que o fluxo volta à ação *Adicionar pizza* e repete o comportamento já descrito, permanecendo nesse laço enquanto houver pizzas para adicionar.

Quando não houver mais pizzas, a atividade encontra um novo nó de decisão que testa se há bebidas associadas ao pedido. Caso não haja, a atividade é encerrada. Caso contrário, é instanciado um novo objeto da classe *Item_Bebida* para cada bebida solicitada no pedido, como demonstra o nó de decisão que teste se ainda há bebidas. Quando não houver mais bebidas, a atividade é encerrada.

Diagrama de Atividade Visualizar Sabores Mais Pedidos

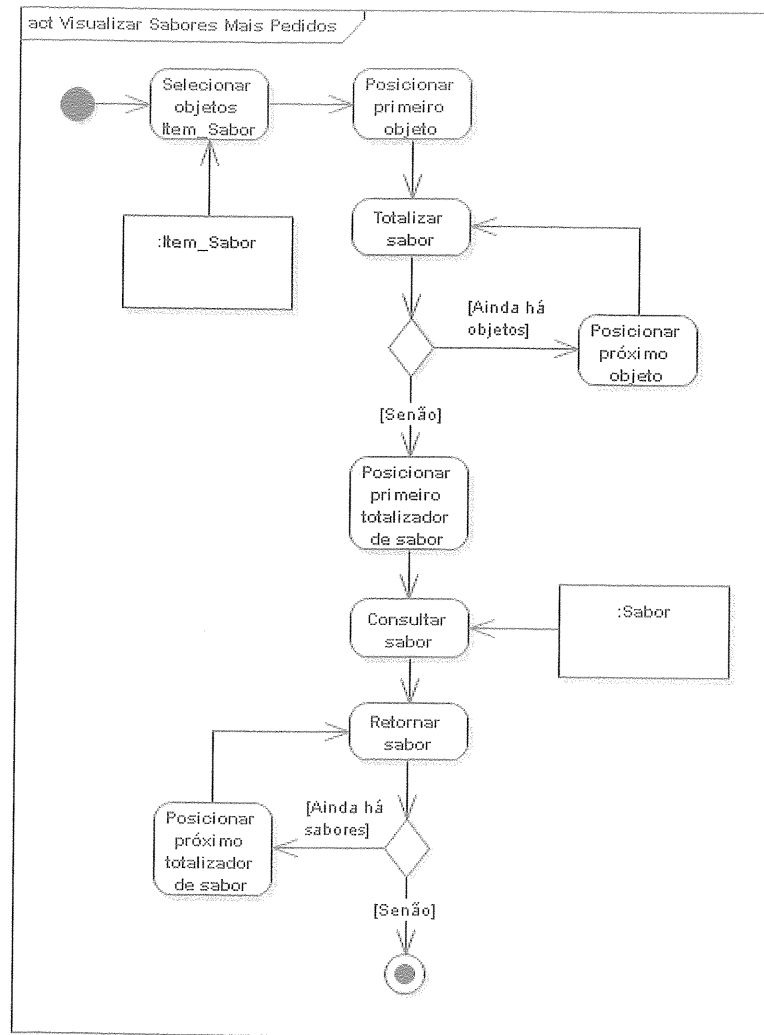


Figura 16.53 – Diagrama de Atividade Visualizar Sabores Mais Pedidos.

Para desenvolver essa atividade, primeiramente é executada a ação que seleciona todos os objetos da classe `Item_Sabor`, conforme demonstra o fluxo de objetos. Em seguida, a atividade posiciona-se sobre o primeiro objeto encontrado, e, na ação seguinte, a ocorrência da solicitação do sabor em questão é somada a um totalizador. Haverá um totalizador para cada sabor.

Em seguida, a atividade encontra um nó de decisão que verifica se ainda há objetos da classe `Item_Sabor`. Em caso positivo, a atividade posiciona sobre o próximo objeto e volta à ação que totaliza os sabores.

Quando não houver mais objetos, a atividade posiciona-se sobre o primeiro totalizador, passando à ação que consulta a descrição do sabor em um objeto da classe `Sabor`, como demonstra o fluxo de objetos. A ação seguinte retorna a descrição do sabor consultado e passa a um nó de decisão, onde é verificado se ainda há totalizadores, caso em que a atividade se posicionará sobre o próximo totalizador e repetirá a consulta da descrição do sabor, conforme já explicado. Quando não houver mais totalizadores a atividade se encerrará.

Diagrama de Atividade Concluir Pedido

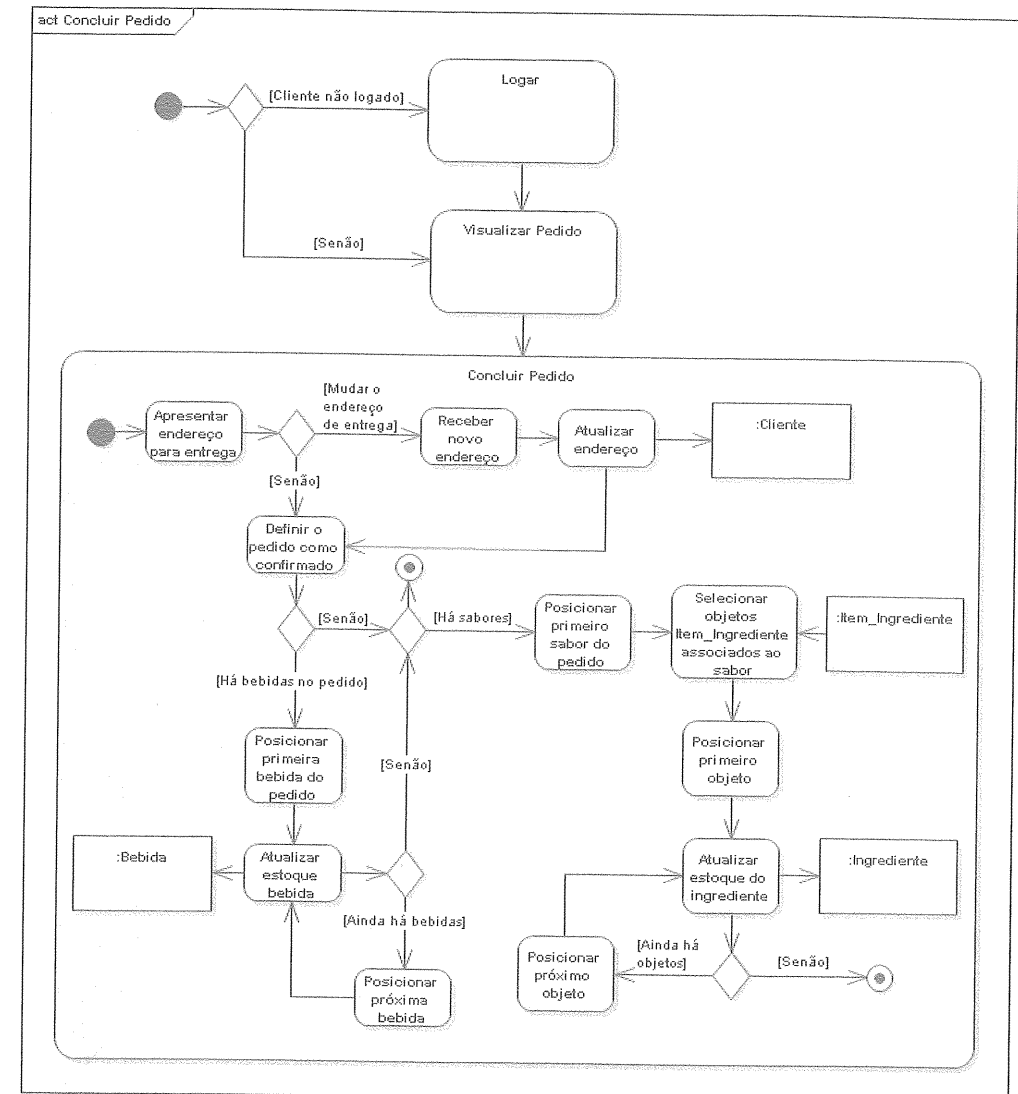


Figura 16.54 – Diagrama de Atividade Concluir Pedido.

Esse diagrama contém três atividades, sendo que as duas primeiras são apenas referenciadas, sem detalhamento de seus nós de ação, enquanto a última apresenta seu detalhamento interno.

O diagrama inicia-se com um nó de decisão, que deve verificar se o cliente ainda não se autenticou, caso em que é necessário executar a atividade de Logar. A seguir, ou caso o cliente já estiver autenticado, passa-se a atividade Visualizar Pedido, na qual será apresentada ainda uma vez os itens contidos no pedido do cliente, esperando uma última confirmação.

A seguir, passa-se a atividade principal desse diagrama, que representa a conclusão do pedido propriamente dita. Essa atividade inicia por apresentar o endereço para entrega do pedido, em seguida a atividade encontra um nó de decisão que verifica se o cliente deseja mudar o endereço de entrega. Se isso for verdadeiro, passa-se a ação que recebe o novo endereço e, a seguir, à ação que atualiza o endereço do cliente. O fluxo de objetos demonstra que a atualização é feita sobre um objeto da classe Cliente.

A seguir, ou caso o cliente não queira modificar seu endereço, passa-se a atividade Definir o pedido como confirmado, que irá alterar a situação do pedido para 1, que, por convenção determina que o pedido foi confirmado pelo cliente.

Depois disso, passa-se a um nó de decisão, no qual se deve verificar se o pedido contém bebidas. Em caso positivo, a atividade se posicionará sobre a primeira bebida do pedido e, em seguida, executará a ação Atualizar estoque bebida, que diminuirá a quantidade solicitada do estoque da bebida, como demonstra o fluxo de objetos.

Em seguida passa-se a um nó de decisão que testa se ainda há bebidas no pedido, caso em que a atividade se posicionará sobre a próxima bebida e repetirá a operação, permanecendo nesse laço até que não haja mais bebidas a atualizar no estoque.

Quando isso ocorrer, ou caso não haja bebidas no pedido, passa-se a um novo nó de decisão em que se deve verificar se há sabores associados ao pedido. Se não houver sabores, a atividade encerra-se nesse ponto, caso contrário, a atividade posiciona-se sobre o primeiro sabor, passando a seguir para a ação que seleciona os objetos da classe Item_Ingrediente associados ao sabor, conforme demonstra o fluxo de objetos entre essa ação e um objeto da classe Item_Ingrediente.

A seguir, a atividade posiciona-se sobre o primeiro desses objetos, passando a seguir para a ação que atualiza o estoque do ingrediente, diminuindo a quantidade necessária para produzir o sabor de sua quantidade em estoque. Isso é demonstrado por meio de um fluxo de objetos.

Logo a seguir, a atividade encontra um nó de decisão, no qual se deve verificar se ainda há objetos da classe Item_Ingrediente, caso em que a atividade deve-se posicionar sobre o objeto seguinte e repetir a atualização do estoque. Quando não houver mais objetos a atividade será encerrada.

Diagrama de Atividade Visualizar Pedidos em Aberto

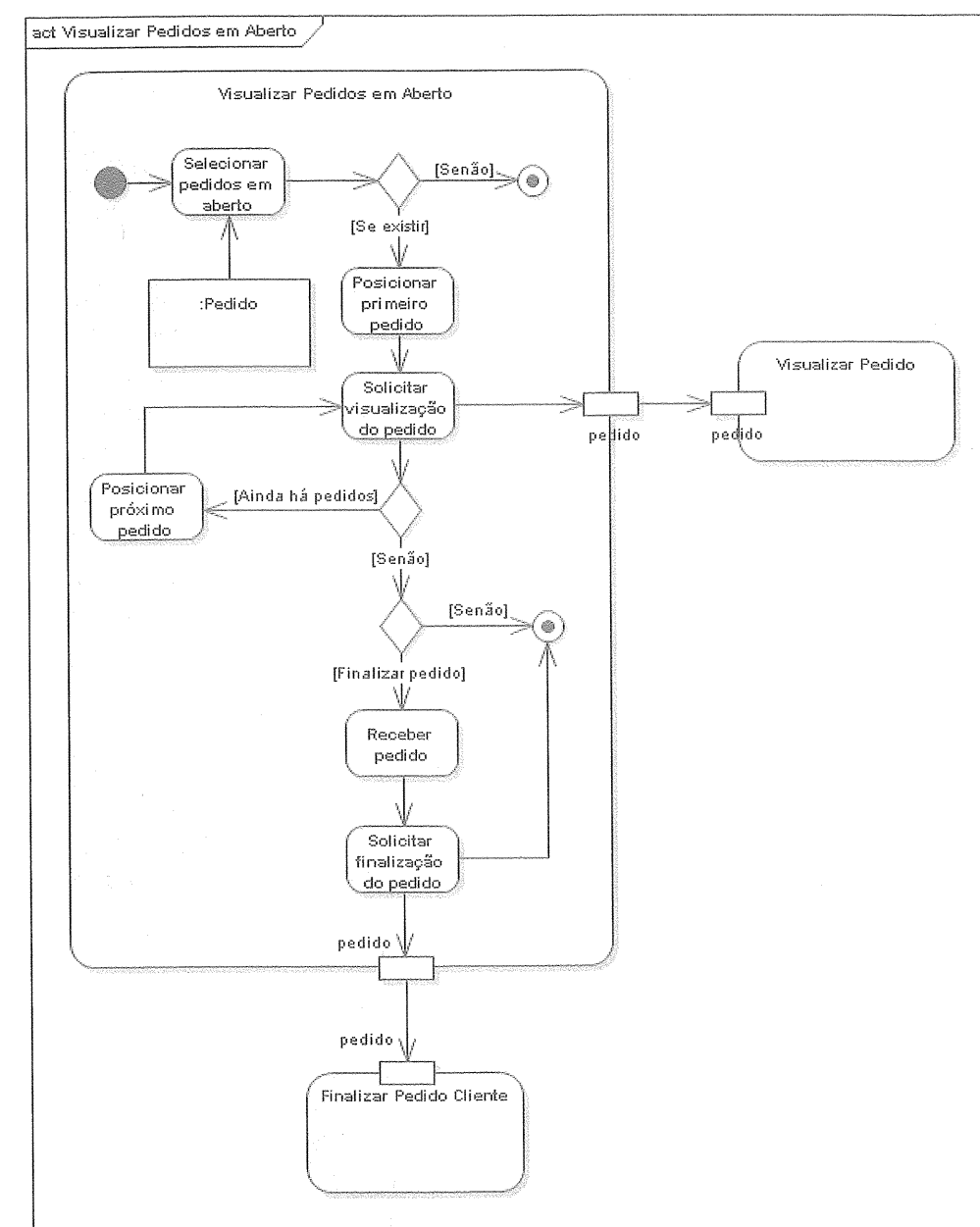


Figura 16.55 – Diagrama de Atividade Visualizar Pedidos em Aberto.

Esse diagrama representa três atividades, sendo que somente a atividade principal detalha seus nós de ação, as outras atividades são apenas referenciadas.

A atividade principal, que se refere ao processo de visualização de pedidos em aberto propriamente dito, inicia-se com a ação *Selecionar pedidos em aberto*, que busca entre os objetos da classe *Pedido*, todos os pedidos cuja situação seja 1, ou seja, que ainda não foram atendidos. O nó de decisão, imediatamente seguinte a esse nó de ação, deve determinar se foram encontrados pedidos em aberto nessa pesquisa. Caso não tenha sido encontrado nenhum pedido nessa situação, a atividade é encerrada.

Se houver pedidos em aberto, a atividade posiciona-se sobre o primeiro objeto encontrado, passando, na ação seguinte, a solicitar sua visualização. Observe que essa solicitação é repassada para a atividade *Visualizar Pedido*, que recebe o número do pedido consultado por meio de nós de parâmetro de atividade.

Em seguida chega-se a outro nó de decisão que verifica se ainda há pedidos a apresentar. Em caso positivo, a atividade posiciona-se sobre o objeto seguinte e repete a solicitação de visualização. Quando não houver mais pedidos, a atividade encontra um novo nó de decisão, que representa a escolha do funcionário entre encerrar o processo ou finalizar um pedido.

Caso o funcionário queira finalizar um pedido, passa-se a ação em que o pedido escolhido pelo funcionário é recebido, e, em seguida, a ação que solicita a finalização do pedido. Essa ação envia o número do pedido a finalizar, por meio de nós de parâmetro de atividade, para a atividade *Finalizar Pedido Cliente*, que se encarregará de finalizá-lo. A seguir, a atividade é encerrada.

Diagrama de Atividade Finalizar Pedido Cliente

Essa atividade inicia-se pela ação que seleciona todos os funcionários registrados na empresa, essa consulta é feita sobre objetos da classe *Funcionario*, como demonstra o fluxo de objetos. Em seguida há um teste, representado por um nó de decisão, que verifica se foram encontrados funcionários. Caso não tenham sido encontrados funcionários, a atividade é encerrada.

Se a pesquisa encontrou funcionários, a atividade posiciona-se sobre o primeiro objeto encontrado, e, na ação seguinte, retorna seu nome para a interface. Em seguida, um nó de decisão verifica se ainda há funcionários a carregar. Em caso positivo, a atividade posiciona-se sobre o objeto seguinte da classe *Funcionario* e volta a retornar seu nome, repetindo essa operação enquanto houver funcionários.

A seguir, a atividade recebe o funcionário que preparou e o funcionário que entregou o pedido, em dois nós de ação diferentes, passando em seguida para o último nó de ação no qual é finalizado o pedido, ou seja, sua situação passa a armazenar o valor 2, que, por convenção determina que o pedido foi atendido. Observe que o objeto da classe *Pedido* é atualizado por meio de um fluxo de objetos.

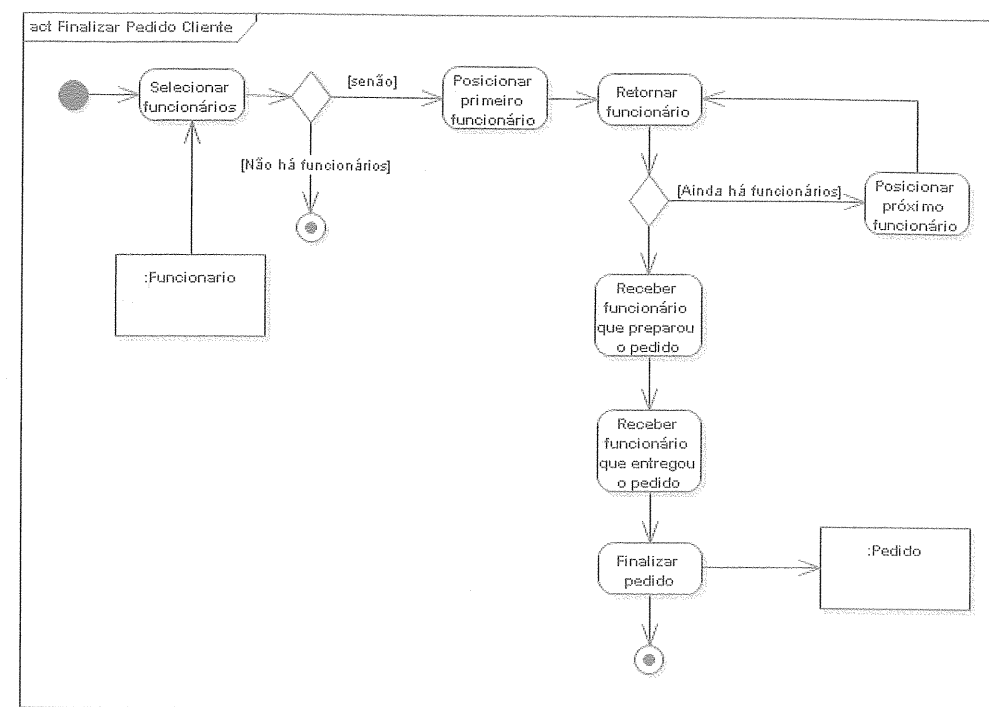


Figura 16.56 – Diagrama de Atividade Finalizar Pedido Cliente.

Diagrama de Atividade Manter Cardápio

Esse diagrama se inicia com a ação que seleciona os sabores registrados no sistema, como demonstra o fluxo de objetos entre essa ação e o objeto da classe *Sabor*, essa consulta seleciona todos os objetos dessa classe. Em seguida, a atividade atinge um nó de decisão que verifica se foi encontrado algum objeto da classe *Sabor*.

Em caso positivo, a atividade posiciona-se sobre o primeiro objeto da classe *Sabor*, e, na ação seguinte, retorna a descrição do sabor para a interface. A seguir, a atividade encontra outro nó de decisão, responsável por determinar se ainda há sabores a carregar, caso em que a atividade se posicionará no objeto seguinte e retornará a descrição do novo sabor, repetindo-se esse laço enquanto houver sabores a apresentar.

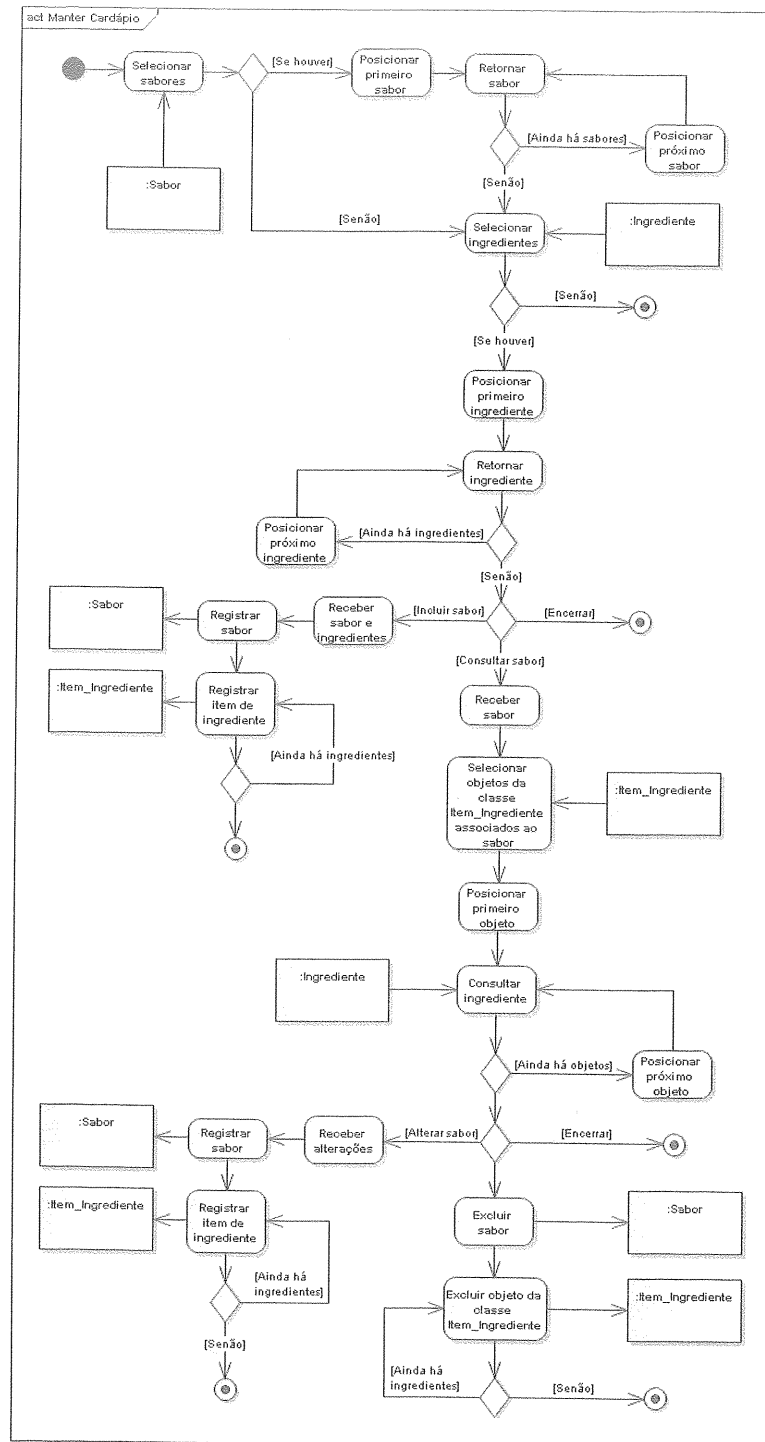


Figura 16.57 – Diagrama de Atividade Manter Cardápio.

Quando não houver mais sabores a apresentar, ou caso não tenha sido encontrado nenhum sabor, passa-se a ação **Selecionar ingredientes**, que seleciona todos os objetos existentes da classe **Ingrediente**, como mostra o fluxo de objetos. Se nenhum ingrediente for encontrado, a atividade é encerrada, como demonstra o nó de decisão seguinte. Caso contrário, a atividade posiciona-se sobre o primeiro ingrediente e, na ação seguinte, o retorna à interface. O próximo nó de decisão informa que essa operação se repetirá enquanto houver ingredientes, com a atividade posicionando-se no ingrediente seguinte, até estes se encerrarem.

A partir daí a atividade encontra um novo nó de decisão, que representa a escolha do administrador entre encerrar o processo, incluir um novo sabor ou consultar um sabor já existente.

Se o administrador quiser inserir um novo sabor, a atividade passa a ação em que recebe o novo sabor e seus ingredientes, por meio da interface. Em seguida passa-se à ação **Registrar sabor**, onde, por meio de um fluxo de objetos, essa ação instancia um novo objeto da classe **Sabor**. A ação seguinte instancia um novo objeto da classe **Item_Ingrediente**, enquanto houver ingredientes para registrar, como demonstra o nó de decisão. Quando não houver mais ingredientes a registrar, a atividade é encerrada.

Caso o administrador queira consultar um sabor já existente, a atividade passa a ação em que recebe o sabor escolhido pelo administrador e, em seguida, executa a ação **Selecionar objetos da classe Item_Ingrediente associados ao sabor**, posicionando-se, em seguida, no primeiro objeto selecionado. A seguir, a ação seguinte retorna a descrição do objeto **Item_Ingrediente**, por meio da consulta ao objeto da classe **Ingrediente** a ele associado. Essa situação se repete enquanto houver objetos da classe **Item_Ingrediente** a apresentar, como demonstra o próximo nó de decisão.

A seguir, a atividade encontra um novo nó de decisão, o qual representa a escolha do administrador entre encerrar o processo, alterar um sabor ou excluir um sabor.

Caso o administrador queira alterar um sabor, procede-se de maneira semelhante à rotina de inserção de um novo sabor. Já se o administrador quiser excluir um sabor, é executada a ação **Excluir sabor**, que destrói o objeto da classe **Sabor** em questão e, em seguida, a atividade inicia um laço em que serão destruídos todos os objetos da classe **Item_Ingrediente** associados ao sabor. Quando não houver mais objetos **Item_Ingrediente** a excluir a atividade será encerrada.

Diagrama de Atividade Emitir Produtos em Baixa no Estoque

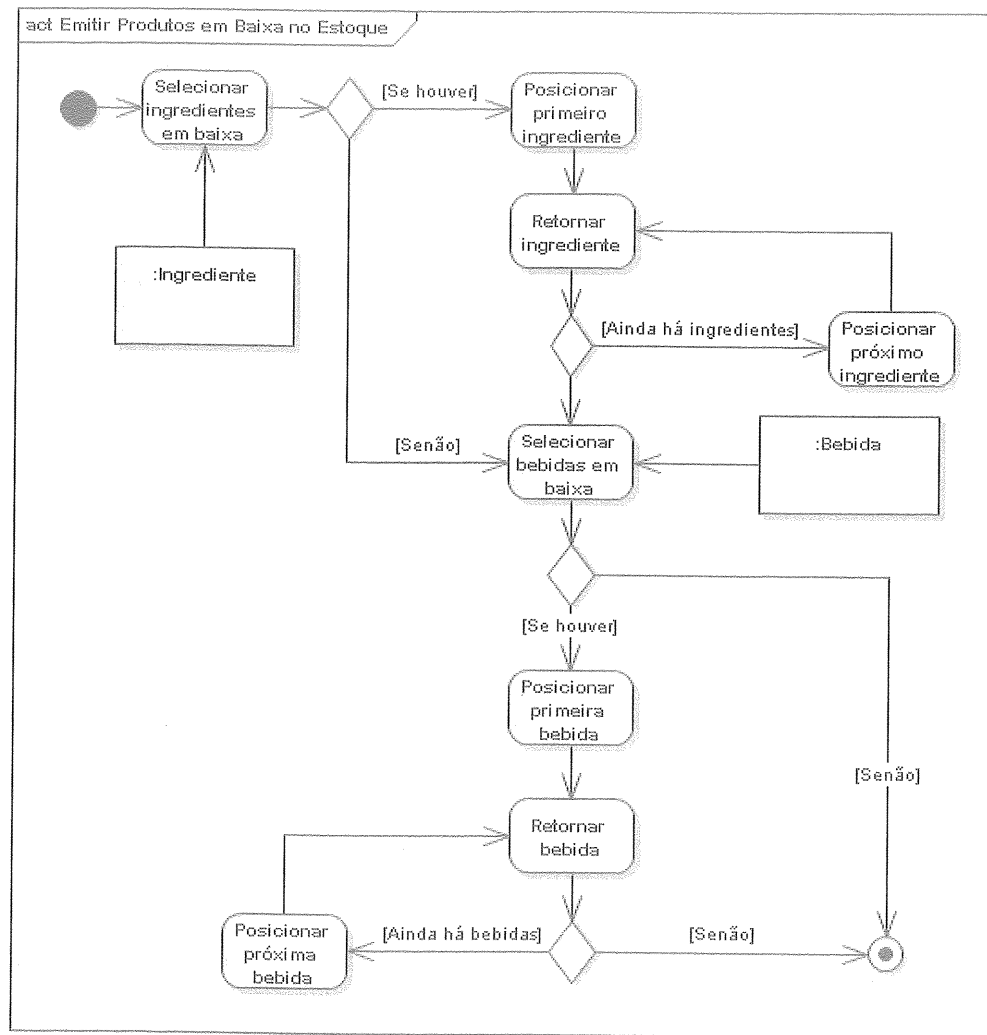


Figura 16.58 – Diagrama de Atividade Emitir Produtos em Baixa no Estoque.

A primeira ação dessa atividade seleciona todos os objetos da classe *Ingrediente* cujas quantidades em estoque estejam abaixo da quantidade mínima. O nó de decisão seguinte a essa ação testa se foi encontrado algum ingrediente em baixa.

Em caso positivo, a atividade posiciona-se sobre o primeiro objeto encontrado e, na ação seguinte, apresenta esse ingrediente. Essa operação se repetirá enquanto houver ingredientes em baixa.

Depois disso, ou caso não tenha sido encontrado nenhum ingrediente em baixa, são selecionadas todas as bebidas em baixa, isto é, todos os objetos da classe *Bebida* cuja quantidade em estoque esteja abaixo da quantidade mínima.

Se não for encontrado nenhum objeto dessa classe com o estoque abaixo do mínimo a atividade é encerrada nesse momento.

Caso contrário, a atividade posiciona-se no primeiro objeto da classe *Bebida* e, na ação seguinte, o retorna, repetindo essa operação enquanto houver objetos da classe *Bebida* para apresentar. Quando estes encerraram-se a atividade também será terminada.

Diagrama de Atividade Emitir Compras em Aberto

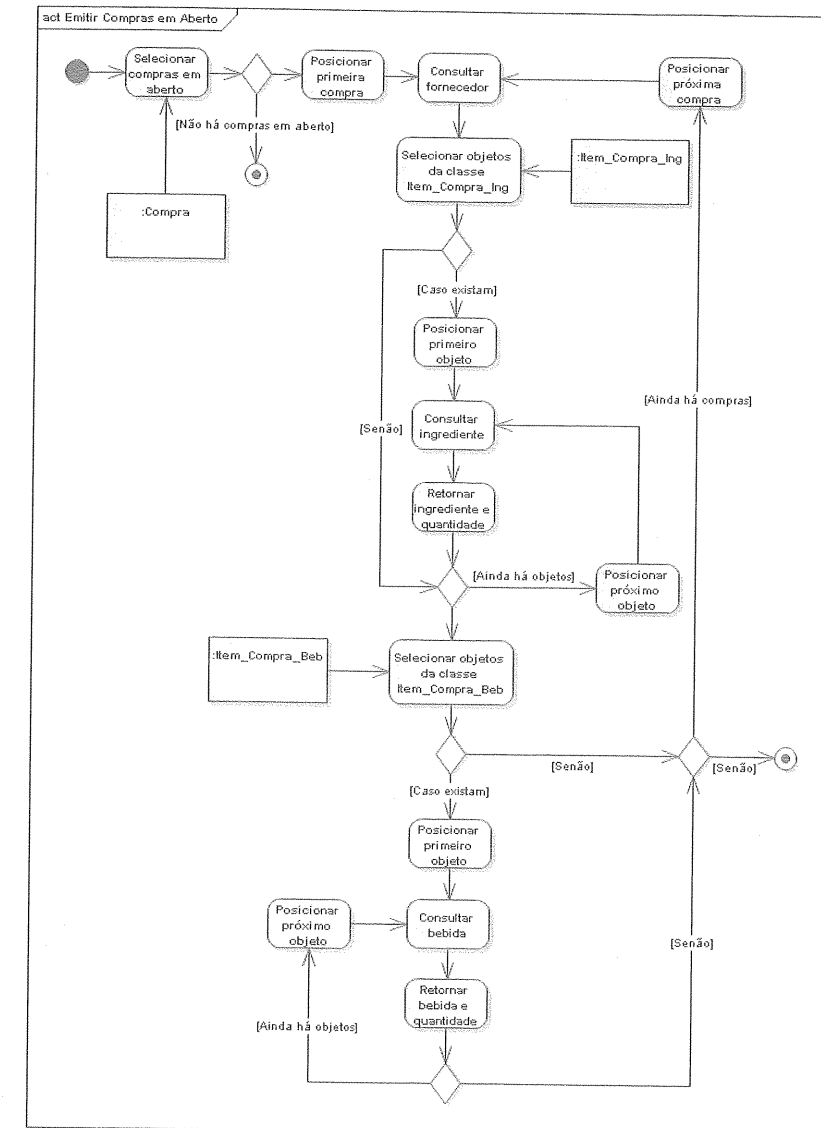


Figura 16.59 – Diagrama de Atividade Emitir Compras em Aberto.

Essa atividade inicia-se com a seleção de todas as compras em aberto, onde é feita uma pesquisa sobre todas as compras cuja situação esteja definida como em aberto, ou seja, ainda não foram entregues. A seguir, é feito um teste para determinar se alguma compra em aberto foi encontrada, em caso negativo a atividade é encerrada.

Caso tenham sido encontradas compras em aberto, a atividade posiciona-se sobre a primeira delas, passando em seguida a consultar o fornecedor associado ao objeto da classe *Compra* em questão. Depois disso, é executada a ação **Selecionar objetos da classe Item_Compra_Ing**, que seleciona todos os objetos dessa classe associados à compra.

Em seguida, um nó de decisão verifica se foram encontrados objetos dessa classe. Em caso positivo, a atividade posiciona-se sobre o primeiro desses objetos, consulta a descrição do ingrediente a ele associado e retorna essa descrição junto com a quantidade solicitada para o ingrediente. Esse comportamento se repetirá enquanto houver objetos da classe *Item_Compra_Ing* a apresentar.

A seguir, ou caso a compra não tenham sido solicitados ingredientes na compra, a atividade seleciona todos os objetos da classe *Item_Compra_Beb* associados à compra. Caso existam objetos dessa classe associados à compra, a atividade posiciona-se sobre o primeiro desses objetos, consulta a descrição da bebida a ele associada e retorna essa descrição junto com a quantidade solicitada. Esse comportamento se repete enquanto houver bebidas associadas à compra.

Quando se esgotarem os objetos da classe *Item_Compra_Beb*, ou caso não tenham sido solicitadas bebidas nessa compra, a atividade encontra um novo nó de decisão, cuja função é verificar se ainda existem compras em aberto a apresentar. Caso existam, a atividade posiciona-se sobre o próximo objeto da classe *Compra* e repete toda a operação já descrita, caso contrário a atividade é encerrada.

Diagrama de Atividade Manter Compras Fornecedor

Esse diagrama contém duas atividades. A primeira atividade representa o processo de **Emitir Compras em Aberto**, descrita na subseção anterior e apenas referenciada aqui.

A segunda atividade se refere ao processo de manutenção de compras propriamente dito, onde, a partir da listagem apresentada na atividade de emissão de compras em aberto, o administrador pode escolher entre finalizar uma compra ou montar um novo pedido, como demonstra o nó de decisão.

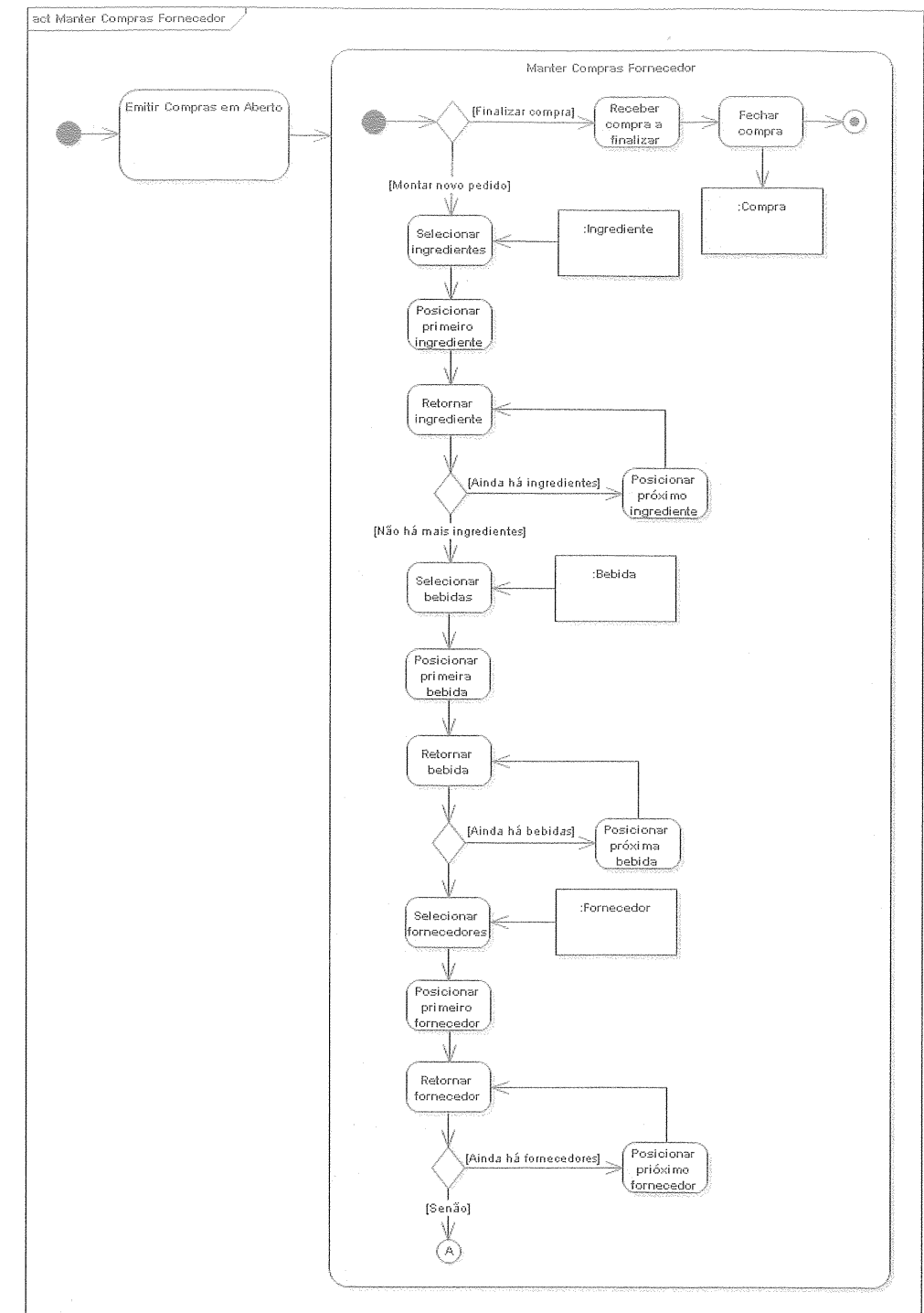


Figura 16.60 – Diagrama de Atividade Manter Compras Fornecedor.

Se o administrador quiser finalizar uma compra, a atividade recebe a compra a finalizar, e, na próxima ação, fecha a compra, definindo a situação do objeto da classe *Compra* como finalizada (definindo o valor do atributo *situacao* como 1). Terminado isso, a atividade é concluída.

No entanto, caso o administrador queira montar uma nova compra, a atividade em resposta seleciona todos os objetos da classe *Ingrediente* registrados, posicionando-se a seguir no primeiro objeto selecionado e retornando a descrição do ingrediente à interface na ação seguinte. Esse comportamento se repetirá enquanto houver ingredientes a carregar na interface.

Quando não houver mais ingredientes, a atividade selecionará todas as bebidas cadastradas no sistema, como demonstra o fluxo de objetos. A seguir, a atividade posiciona-se sobre o primeiro objeto selecionado e, na ação seguinte, retorna a descrição da bebida para a interface. Esse laço se repetirá enquanto houver bebidas a apresentar na interface.

Depois disso, a atividade selecionará todos os fornecedores registrados no sistema, posicionando-se em seguida sobre o primeiro objeto da classe *Fornecedor* encontrado, retornando seu nome à interface na ação seguinte. Novamente a atividade permanecerá executando esse comportamento enquanto houver fornecedores a carregar na interface.

Observe que, quando não houver mais fornecedores, a atividade atinge um conector, que representa um atalho para a continuação desse diagrama, apresentando na figura seguinte. Isso foi necessário porque esse diagrama é muito extenso e tivemos, portanto que dividi-lo em duas figuras.

Na continuação dessa atividade, a ação seguinte recebe o fornecedor escolhido pelo administrador e a próxima ação recebe os produtos a comprar. Em seguida a compra é registrada, gerando um novo objeto da classe *Compra*, como demonstra o fluxo de objetos.

Em seguida a atividade encontra um nó de decisão, que deve verificar se o administrador escolheu ingredientes para comprar. Nesse caso a atividade posiciona-se no primeiro ingrediente, registra-o, gerando um novo objeto da classe *Item_Compra_Ing* e permanece nessa operação enquanto houver ingredientes a registrar.

A seguir, ou caso não tenham sido solicitados ingredientes na compra, a atividade verifica, por meio de um nó de decisão, se foram solicitadas bebidas na compra. Em caso negativo a atividade encerra-se nesse momento, caso contrário, a atividade posiciona-se sobre a primeira bebida solicitada, registra-a, gerando um novo objeto da classe *Item_Compra_Beb* e permanece nesse laço enquanto houver bebidas a registrar. Quando estes acabarem a atividade será encerrada.

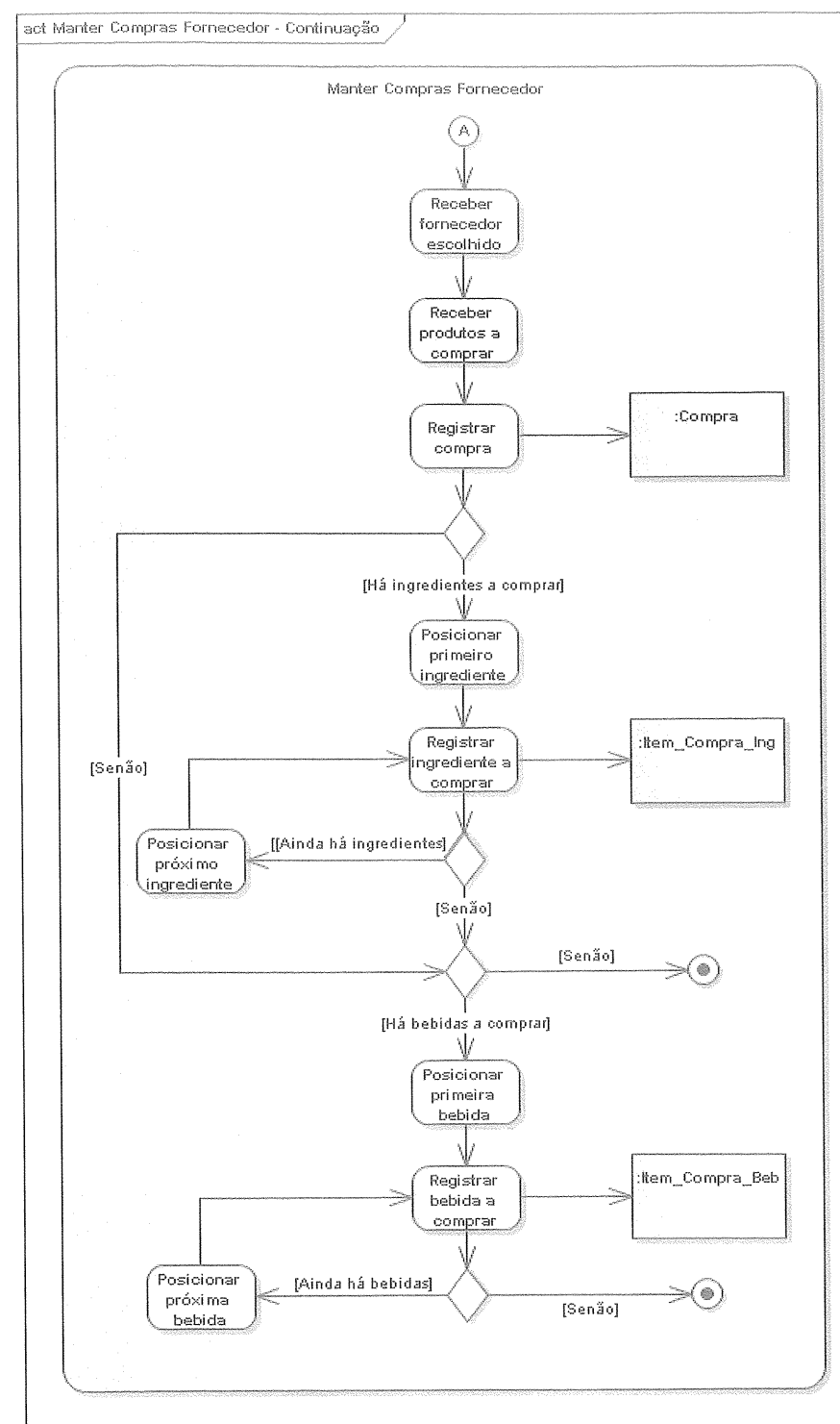


Figura 16.61 – Diagrama de Atividade Manter Compras Fornecedor – Continuação.

Diagrama de Atividade Emitir Melhores Clientes

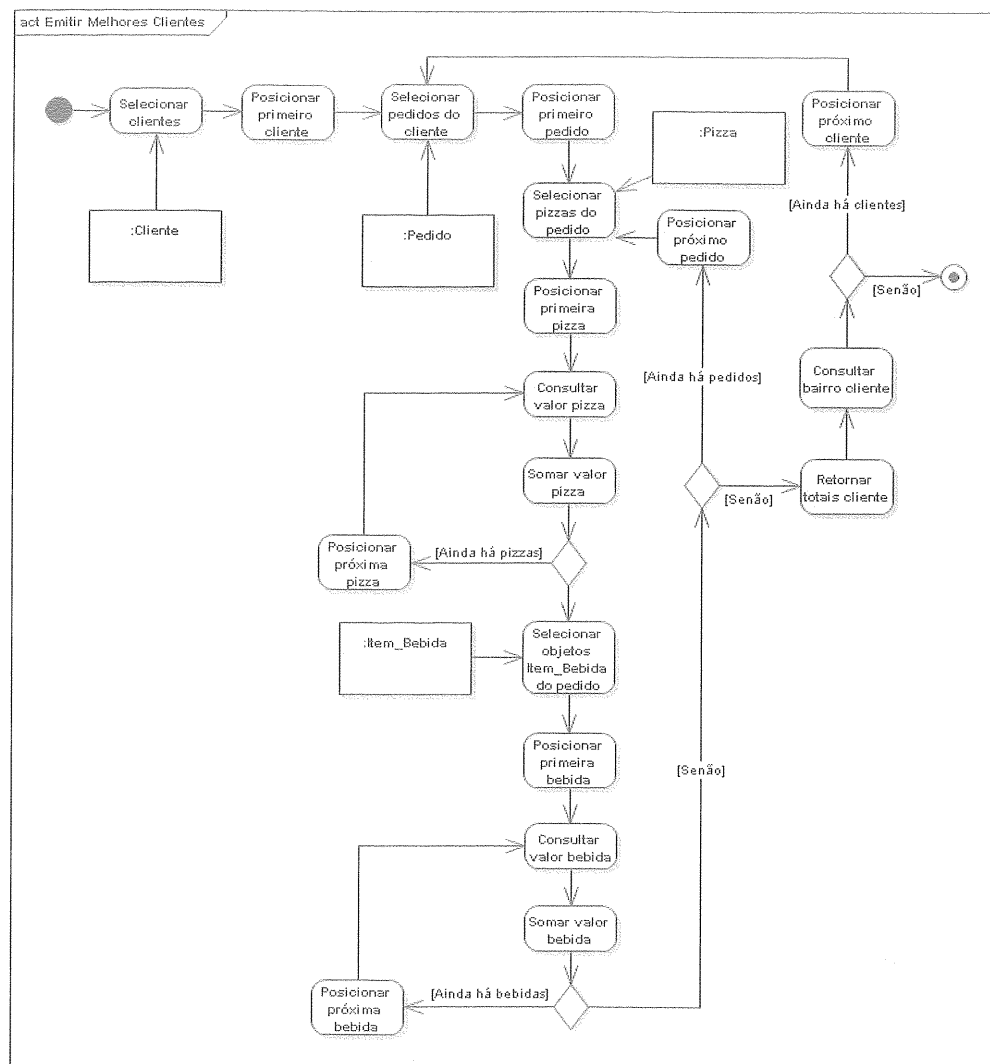


Figura 16.62 – Diagrama de Atividade Emitir Melhores Clientes.

A solicitação dessa listagem obriga a atividade a selecionar todos os clientes do sistema, posicionando-se sobre o primeiro e selecionando todos os pedidos já realizados pelo mesmo. Em seguida, a atividade irá se posicionar no primeiro pedido do cliente e selecionará todas as pizzas solicitadas nele, posicionando-se na primeira pizza, consultando seu valor e somando este a um totalizador dos gastos do cliente. A seguir, se ainda houver pizzas, a atividade se posicionará na pizza seguinte e repetirá a operação anterior.

Quando não houver mais pizzas, a atividade selecionará todos os objetos Item_Bebida associados ao pedido, posicionando-se no primeiro, consultando seu valor e somando-o ao totalizador do cliente. Esse comportamento se repetirá enquanto houver bebidas a totalizar.

No momento em que não houver mais bebidas, a atividade verificará se ainda existem pedidos do cliente atual. Se ainda existir pedidos, a atividade se posicionará sobre o próximo pedido e repetirá a rotina já descrita.

Se não existirem mais pedidos, a atividade retornará os totais do cliente, consultará seu bairro e tentará se posicionar no próximo cliente, se houver. Se ainda existirem clientes, toda a operação já descrita será repetida. Caso contrário, o sistema apresentará a listagem de clientes em ordem decrescente de seus totalizadores e o processo será encerrado.

Diagrama de Atividade Emitir Consumo por Período

A primeira ação dessa atividade constitui-se em receber os períodos desejados para a listagem. A seguir, a atividade seleciona todos os objetos da classe Pedido que se encontrem dentro do período informado. Se não for encontrado nenhum pedido nesse período a atividade é encerrada.

Caso contrário, a atividade posiciona-se sobre o primeiro objeto Pedido encontrado, seleciona todos os objetos da classe Pizza a ele associados, posiciona-se sobre o primeiro objeto Pizza e consulta o número de pedaços da pizza, por meio de uma consulta a um objeto da classe Tamanho associado ao objeto da classe Pizza.

Depois disso, são selecionados todos os objetos da classe Item_Sabor associados ao objeto Pizza, posiciona-se sobre o primeiro objeto e consulta-se o objeto da classe Sabor a ele associado.

Em seguida a atividade precisa selecionar todos os objetos da classe Item_Ingrediente associados ao objeto Sabor, posicionar no primeiro objeto encontrado e somar a quantidade necessária do ingrediente para produzir o sabor em um totalizador. A seguir, é consultado a descrição do objeto da classe Ingrediente associado ao objeto Item_Ingrediente, atingindo-se em seguida um nó de decisão que verifica se ainda há objetos da classe Item_Ingrediente. Se houver a atividade posiciona-se no próximo objeto e repete a operação.

Caso não haja mais objetos da classe Item_Ingrediente, a atividade testa se ainda há objetos da classe Item_Sabor associados ao objeto Pizza. Em caso positivo, a atividade posiciona-se sobre o próximo objeto Item_Sabor e repete a rotina de consultar o objeto Sabor associado ao objeto Item_Sabor e todo o restante da rotina.

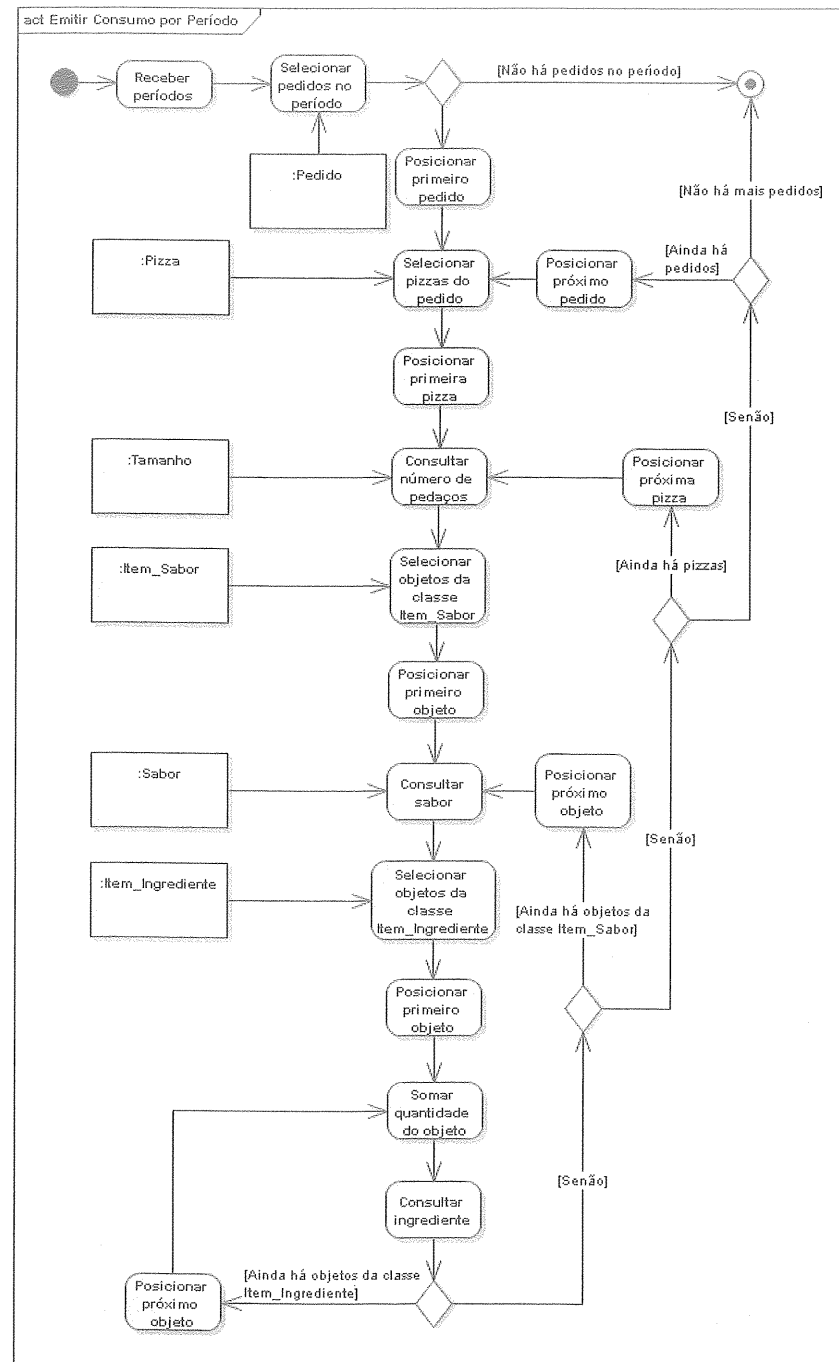


Figura 16.63 – Diagrama de Atividade Emitir Consumo por Período.

Se não houver mais objetos Item_Sabor, a atividade verifica se ainda há objetos Pizza associados ao objeto Pedido. Em caso positivo a atividade posiciona-se sobre

o próximo objeto Pizza e volta a consultar o número de pedaços do objeto Tamanho associado ao objeto Pizza, repetindo toda a operação já descrita a partir daí.

Se não houver mais objetos Pizza, a atividade deve determinar se ainda há objetos da classe Pedido. Se ainda houver objetos Pedido, a atividade posiciona-se sobre o próximo objeto e seleciona os objetos da classe Pizza associados ao objeto Pedido, repetindo todo o algoritmo já descrito a partir desse momento. Caso não hajam mais pedidos, a atividade é encerrada.

16.2.10 Diagrama de Visão Geral de Interação – Realizar Pedido

Nesta seção modelaremos o processo geral para que um cliente realize um pedido por meio de um diagrama de visão geral de interação, apresentado na figura 16.64. O leitor notará que utilizamos somente quadros de ocorrência de interação, pois não havia espaço para inserir quadros de interação completos.

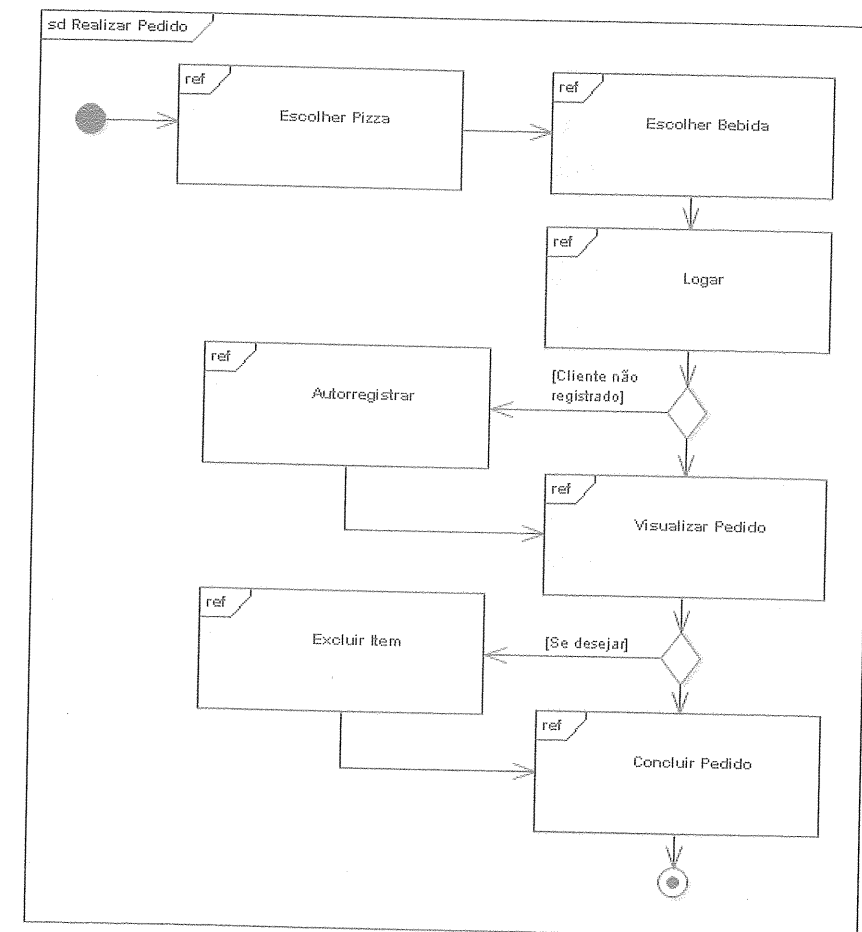


Figura 16.64 – Diagrama de Visão Geral de Interação – Realizar Pedido.

Esse diagrama é fácil de interpretar, representando todos os processos necessários para que um cliente realize um pedido. Primeiramente é executado o processo de **Escolher Pizza**, passando para o processo de **Escolher Bebida**. Depois disso, o cliente necessita se autenticar no sistema, por meio do processo **Logar** e, caso não esteja registrado no sistema, deverá se autorregistrar, como demonstra o nó de decisão seguinte ao processo **Logar**.

Após estar autenticado, é necessário visualizar o pedido (o cliente pode fazer isso a qualquer momento, no entanto, somente para concluir o pedido isso é obrigatório). Durante a visualização do pedido, o cliente pode, se assim o desejar, excluir algum item do pedido. Isso é representado pelo nó de decisão seguinte ao processo de **Visualizar Pedido**, definindo que, se o cliente quiser, será executado o processo de **Excluir Item**.

Finalmente, é executado o processo de **Concluir Pedido**, onde o pedido será definido como concluído e será dada baixa nos itens necessários para atendê-lo.

16.2.11 Diagrama de Componentes da PizzaNet

Nesta seção iremos modelar o diagrama de componentes da PizzaNet. Nesse diagrama os componentes aqui modelados referem-se aos módulos executáveis do sistema, bem como os módulos necessários para o seu funcionamento.

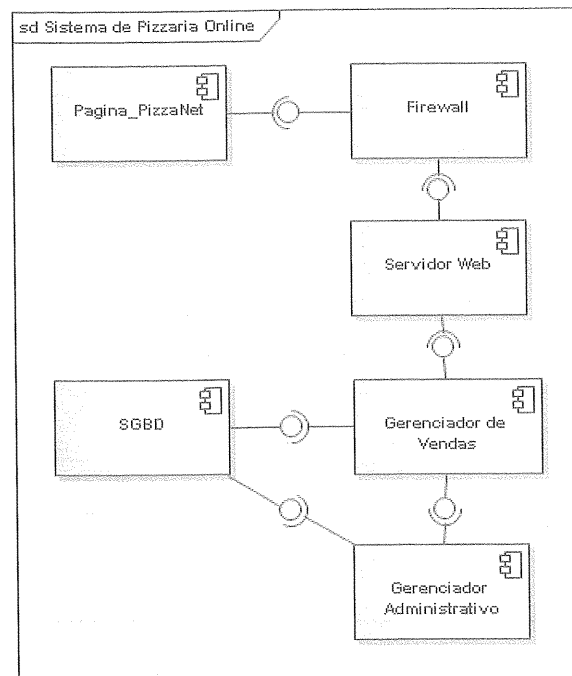


Figura 16.65 – Diagrama de Componentes da PizzaNet.

Como o leitor pode observar, esse diagrama é formado pelos seguintes componentes:

- **Pagina_PizzaNet** – representa a página da pizzaria, que é carregada e atualizada na máquina do cliente e por meio da qual estes solicitam pedidos à PizzaNet.
- **Firewall** – representa um software de firewall que recebe o tráfego de informações externas ao sistema, incluindo os eventos que ocorrem nas páginas dos clientes da pizzaria. Sua função é impedir a entrada de usuários, softwares ou informações não autorizados na rede interna da empresa. Esse componente não faz realmente parte do sistema, mas é indispensável para o seu bom funcionamento.
- **Servidor Web** – representa um software responsável, entre outras coisas, por gerenciar os múltiplos pedidos de conexão solicitados pelos clientes da PizzaNet. Da mesma forma que o componente anterior, não faz realmente parte do sistema, mas é necessário a ele.
- **Gerenciador de Vendas** – representa o módulo que atende as solicitações dos clientes, como escolha de pizzas, de bebidas, visualização do pedido, conclusão do pedido etc. Em suma este é o componente que implementa todas as funcionalidades oferecidas aos clientes da pizzaria, representando o subsistema de vendas definido no diagrama de pacotes deste capítulo. Ele poderia ser dividido em diversos módulos, mas acreditamos que um único componente poderia desempenhar essa função.
- **Gerenciador Administrativo** – engloba todos os serviços oferecidos aos funcionários da empresa, como finalizar um pedido, solicitar compras de produtos ou emitir os melhores clientes. Ele representa o subsistema administrativo. Consideramos que apenas um componente seria capaz de gerenciar todas essas funcionalidades, embora nada impedisse que essas fossem divididas em mais de um componente.
- **SGBD** – finalmente, esse componente representa um Sistema Gerenciador de Banco de Dados responsável por armazenar e recuperar as informações necessárias ao sistema.

16.2.12 Diagrama de Implantação da PizzaNet

Nesta seção iremos concluir a modelagem do sistema de pizzaria online, por meio da modelagem do diagrama de implantação da PizzaNet, onde demonstraremos as necessidades físicas para que o sistema possa ser implantado. Nesse diagrama, os componentes modelados na seção anterior são manifestados por artefatos, contidos pelos nós que os suportarão.

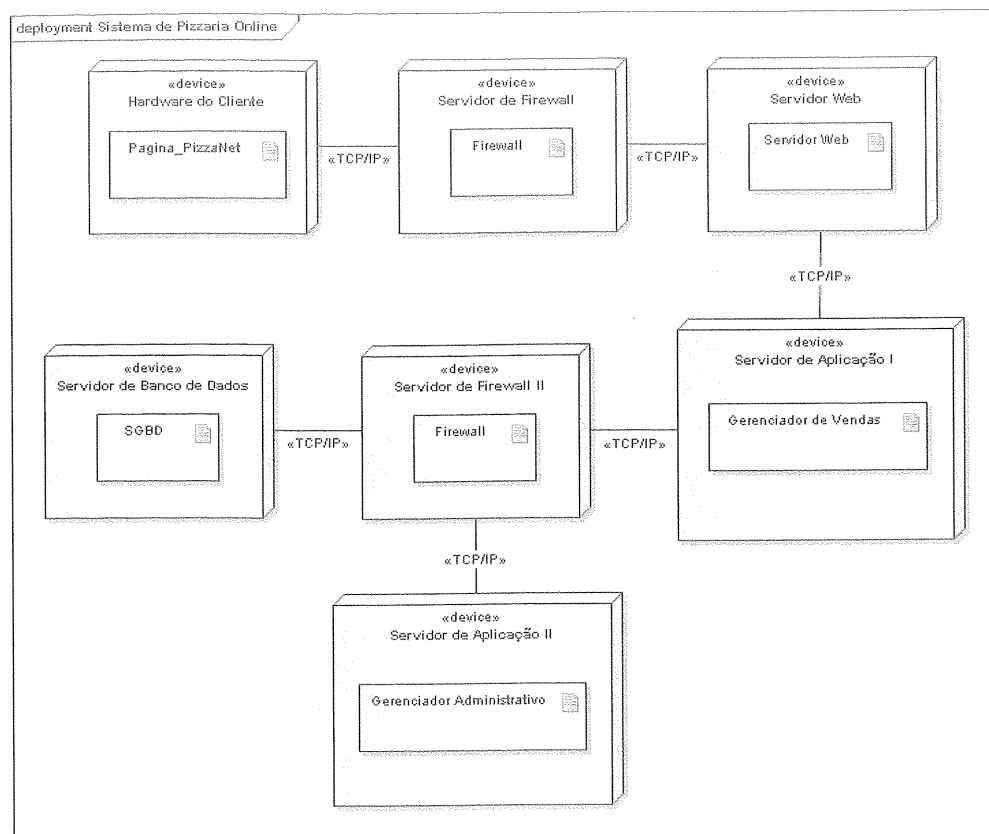


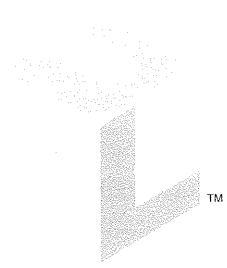
Figura 16.66 – Diagrama de Implantação da PizzaNet.

A seguir descreveremos cada um dos nós que compõem esse diagrama:

- **Hardware do Cliente** – representa as máquinas dos clientes, sobre o qual será carregada a página da PizzaNet, como demonstra o artefato *Pagina_PizzaNet*, que é uma manifestação do componente de mesmo nome.
- **Servidor de Firewall** – representa a máquina onde irá rodar um software de firewall, que procurará impedir invasões de pessoas não autorizadas à rede interna da pizzeria.
- **Servidor Web** – representa o hardware necessário para suportar um software que deverá gerenciar os múltiplos pedidos de conexão solicitados pelos clientes da PizzaNet.
- **Servidor de Aplicação I** – representa o servidor que irá suportar o subsistema de vendas da PizzaNet, representado pelo artefato *Gerenciador de Vendas*.

- **Servidor de Firewall II** – este é um servidor que deve suportar um segundo software de firewall, fornecendo uma segurança extra aos dois servidores mais internos da rede da PizzaNet.
- **Servidor de Banco de Dados** – representa um servidor que deverá executar um sistema gerenciador de banco de dados, que deverá armazenar e recuperar as informações necessárias ao sistema.
- **Servidor de Aplicação II** – representa o último servidor da rede, onde é executado o subsistema administrativo da PizzaNet, representado pelo artefato *Gerenciador Administrativo*.

Obviamente em um sistema relativamente simples como este, dificilmente seriam necessários tantos servidores, muitos dos módulos aqui representados poderiam estar contidos em uma única máquina. Na verdade, a rigor, o sistema todo poderia ser executado em um único servidor. Criamos esse diagrama mais complexo para ilustrar como um sistema poderia ser distribuído entre vários servidores.



CAPÍTULO 17

A UML 2

Os objetivos que levaram os desenvolvedores da linguagem UML a lançar a versão 2 foi reestruturá-la e refiná-la de maneira a torná-la mais fácil de aplicar, implementar e adaptar, melhorando sua precisão e capacidade de reutilização.

Um dos principais problemas enfrentados pela linguagem UML era causado pela constante evolução tecnológica na área de software, o que impedia que a UML se tornasse totalmente utilizável no momento em que uma empresa adotava uma nova tecnologia, devido ao fato de a UML não prever ou não ser compatível com determinadas características inovadoras da tecnologia em questão. Assim, os desenvolvedores da UML decidiram permitir a criação de perfis a partir do núcleo da linguagem, permitindo, dessa maneira, que os usuários adaptassem a linguagem a uma plataforma ou a um domínio de aplicação específico.

17.1 Conceitos Básicos: Modelos, Metamodelos e Metaclasses

Aqui iremos fornecer algumas definições relativas a termos que serão amplamente usados nas seções seguintes.

Um modelo captura uma visão de um sistema físico, é uma abstração do sistema com certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que é irrelevante. Assim o modelo descreve completamente aqueles aspectos do sistema físico que são relevantes ao propósito do modelo, no nível apropriado de detalhe.

Um metamodelo é um modelo que define uma linguagem para expressar modelos. O papel de um metamodelo é o de definir a semântica para modelar elementos dentro de um modelo sendo instanciado. Dessa forma um modelo é uma instância de um metamodelo.

Uma metaclasses é uma classe, em um metamodelo, cujas instâncias são elementos concretos da UML, como classes, casos de uso ou atores. Metaclasses são usadas para construir metamodelos.

17.1.1 Metaclasses Utilizadas para a Modelagem de Casos de Uso em UML

Como uma forma de ilustrar os conceitos de metaclasses e metamodelos, apresentamos na figura 17.1 o metamodelo contendo as metaclasses que representam os conceitos utilizados para a modelagem de casos de uso.

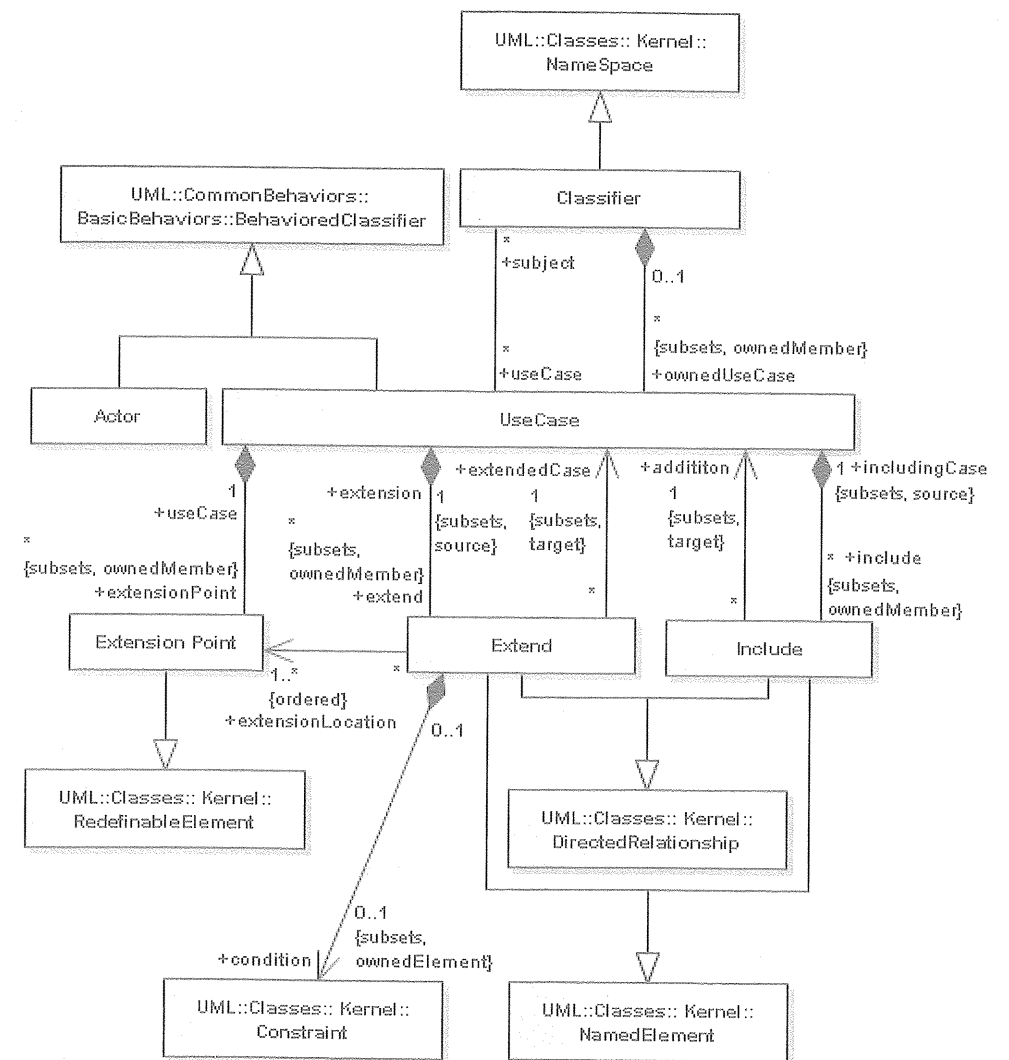


Figura 17.1 – Metamodelo contendo as Metaclasses Utilizadas para a Modelagem de Casos de Uso.

Ao observarmos essa figura, percebemos que as metaclasses centrais, Actor, UseCase, Extension Point, Extend e Include, correspondem aos conceitos já explicados no capítulo 3 sobre atores, casos de uso, pontos de extensão e as associações de extensão e inclusão. Assim, um ator representado em um modelo nada mais é que uma instância da metaclasses Actor. A seguir explicaremos sucintamente as outras metaclasses contidas nesse metamodelo, e, quando necessário, os pacotes de onde foram derivadas.

17.1.2 Metaclasses Classifier

Um classificador (classifier) é uma classificação de instâncias, ele descreve um conjunto de instâncias que têm características em comum. É um espaço de nome cujos membros podem incluir características. É uma metaclasses abstrata. Um classificador é um tipo e pode ter generalizações, tornando possível definir relacionamentos de generalização para outros classificadores. Pode especificar uma hierarquia de generalização por meio de referência aos seus classificadores gerais. É um elemento redefinível, o que significa que é possível redefinir classificadores aninhados.

Um classificador é um mecanismo que descreve características comportamentais e estruturais, porém as instâncias de um classificador não são objetos como as instâncias de uma classe, mas elementos UML concretos (que incluem classes). Ele não pode ser utilizado em um modelo, somente suas instâncias o podem sê-lo. Classificadores são utilizados apenas em metamodelos.

17.1.3 Pacote CommonBehaviors, Subpacote BasicBehaviors e Metaclasses BehavioredClassifier

O pacote CommonBehaviors especifica os conceitos centrais requeridos para elementos dinâmicos e fornece a infraestrutura para suportar definições de comportamento mais detalhadas.

O subpacote BasicBehaviors introduz o arcabouço (framework) que será usado para especificar comportamentos, enquanto os subtipos concretos de Comportamento (Behavior) fornecerão mecanismos diferentes para especificar comportamentos.

O BehavioredClassifier é um classificador que tem especificações de comportamento definidas em seu namespace.

Observe a descrição da metaclasses "UML::CommonBehaviors::BasicBehaviors::BehavioredClassifier. Esses dois pontos seguidos (::) significam que a metaclasses BehavioredClassifier está contida no subpacote BasicBehaviors que está contido no pacote CommonBehaviors que por sua vez está contido no pacote UML.

Observe ainda que é a partir da metaclasses BehavioredClassifier que são derivadas as metaclasses Actor e UseCase.

17.1.4 Pacotes Classes e Kernel e Metaclasses Namespace, NamedElement, DirectedRelationship, Constraint e RedefinableElement

O pacote Classes contém subpacotes que lidam com conceitos básicos de modelagem da UML, em particular classes e seus relacionamentos.

O pacote Kernel representa os conceitos de modelagem centrais da UML, incluindo classes, associações e pacotes.

A metaclasses Namespace (espaço de nome) representa um elemento em um modelo que contém um conjunto de elementos nomeados que podem ser identificados pelo nome. É uma metaclasses abstrata. Observe ainda que a metaclasses Classifier foi derivada a partir da metaclasses Namespace. Observe que essa metaclasses está contida no pacote Kernel, que está contido no pacote Classes contido por sua vez no pacote UML, da mesma forma que as metaclasses que explicaremos a seguir.

Um Named Element (elemento nomeado) é uma metaclasses que representa elementos que podem ter um nome. O nome é usado para identificação do elemento nomeado dentro do espaço de nome no qual está definido. É uma metaclasses abstrata, contida no pacote Kernel a partir da qual foram especializadas as metaclasses Extend e Include.

Um Directed Relationship (relacionamento direcionado) é uma metaclasses que representa um relacionamento entre uma coleção de elementos de modelo fonte e uma coleção de elementos de modelo destino. É uma metaclasses abstrata contida no pacote Kernel que também foi utilizada para especializar as metaclasses Extend e Include, denotando uma herança múltipla.

Uma Constraint (restrição) é uma metaclasses que representa uma condição ou restrição expressada em texto em linguagem natural ou linguagem de programação para o propósito de declarar algumas das semânticas de um elemento. Constraint é uma condição que restringe a extensão do elemento associado além do que é imposto por outras construções de linguagem aplicadas naquele elemento. É uma metaclasses abstrata associada à metaclasses Extend, uma vez que esta é uma associação que necessita que uma condição seja satisfeita..

Um Redefinable Element (elemento redefinível) é uma metaclasses que representa um elemento nomeado que pode ser redefinido no contexto de uma generalização. É uma metaclasses abstrata, contida no pacote Kernel, a partir da qual foi derivada a metaclasses Extension Point.

17.2 A Arquitetura da Linguagem

A especificação da linguagem UML 2 é definida por meio da utilização de uma abordagem de metamodelagem que adapta técnicas de especificação formal. Embora essa abordagem necessite de um pouco do rigor de um método de especificação formal, ela oferece as vantagens de ser mais intuitiva e pragmática.

17.2.1 Princípios de Projeto da UML 2

O metamodelo UML foi desenvolvido com o objetivo de abranger os princípios de projeto apresentados a seguir:

- **Modularidade** – é aplicado a grupos de construções dentro de pacotes e organiza características em metaclasses.
- **Divisão por camadas (Layering)** – é aplicado de duas formas ao metamodelo UML. Primeiro, a estrutura do pacote é dividida em camadas para separar as construções do núcleo da metalinguagem das construções de nível mais alto que utilizam essas construções. Segundo, um metamodelo de quatro camadas, padrão arquitetural, é consistentemente aplicado para separar interesses por meio das camadas de abstração.
- **Particionamento** – é usado para organizar áreas conceituais dentro da mesma camada. No caso da Biblioteca de Infraestrutura, um particionamento bem-definido é usado para fornecer a flexibilidade requerida pelos padrões de metamodelagem atuais e futuros. No caso do metamodelo UML, o particionamento é fracamente definido de maneira a melhorar a coesão dentro dos pacotes e diminuir o acoplamento entre os mesmos.
- **Extensibilidade** – a UML pode ser estendida de duas maneiras. Na primeira um novo dialeto da UML pode ser definido pelo uso de Perfis (Profiles) para customizar a linguagem para plataformas e domínios particulares, enquanto na segunda, uma nova linguagem relacionada à UML pode ser especificada pela reutilização de parte do pacote da Biblioteca de infraestrutura e aumentada com metaclasses e metarelacionamentos apropriados.
- **Reuso** – uma biblioteca de metamodelo flexível e bem-definida é fornecida de maneira que possa ser reutilizada para definir o metamodelo da UML, além de outros metamodelos relacionados arquiteturalmente, tal como o Meta Object Facility (MOF – Instalação para Meta Objeto) e o Common Warehouse Model (CWM – Modelo para Depósito Comum).

O Meta-Object Facility (MOF) é a tecnologia adotada pela OMG (Object Management Group – Grupo de Gerenciamento de Objeto) para definir metadados e representá-los como objetos CORBA. Um metamodelo MOF define a sintaxe abstrata dos metadados na representação MOF de um modelo. O próprio modelo MOF descreve a sintaxe abstrata para representar metamodelos MOF que podem ser representados usando um subconjunto da sintaxe UML.

O modelo MOF é formado por dois pacotes principais, EMOF (Essential MOF – MOF Essencial) e CMOF (Complete MOF – MOF Completa). O modelo CMOF é utilizado para especificar outros metamodelos, como a UML 2. No entanto, é construído a partir do núcleo da UML.

17.2.2 A Infraestrutura da UML 2

O objetivo da infraestrutura da UML é definir um núcleo de metalinguagem que possa ser utilizado para definir a UML e possa ser reutilizado para definir outras arquiteturas de metamodelos, além de definir mecanismos que permitam a personalização e a adaptação da UML, de maneira a criar “dialeto” para plataformas e domínios específicos.

A infraestrutura da UML define as construções básicas de especificação da UML, além de permitir adaptá-la a domínios específicos. A infraestrutura da UML é definida pela Biblioteca de Infraestrutura, que satisfaz os seguintes requisitos de projeto:

- Definir um núcleo de metalinguagem que possa ser reutilizada para definir uma variedade de metamodelos, incluindo UML, MOF e CWM.
- Alinhar arquiteturalmente UML, MOF e XMI de forma que o intercâmbio entre os modelos seja totalmente suportado.
- Permitir personalização da UML por intermédio de perfis (Profiles) e a criação de novas linguagens (famílias de linguagens) baseadas no mesmo núcleo de metalinguagem que a UML.

A figura 17.2 mostra como estão organizados os pacotes que compõem o pacote da Biblioteca de Infraestrutura da UML.

Nessa figura percebemos que o pacote da Biblioteca de Infraestrutura subdivide-se nos pacotes Núcleo e Perfis. Observe que o pacote Perfis tem uma associação de dependência com o pacote Núcleo (Core), uma vez que um perfil é sempre derivado a partir do núcleo.

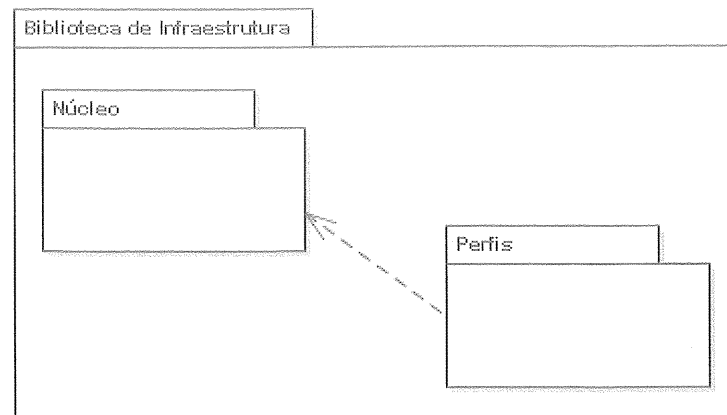


Figura 17.2 – Pacotes da Biblioteca de Infraestrutura da UML.

17.2.3 O Pacote Núcleo da Biblioteca de Infraestrutura

Em sua primeira funcionalidade, o pacote Núcleo (Core) é um metamodelo completo particularmente projetado para alta reusabilidade, enquanto outros metamodelos no mesmo metanível importam ou especializam suas metaclasses específicas. Uma vez que tais metamodelos estão no centro da MDA (Model Driven Architecture – Arquitetura Orientada ao Modelo), o núcleo comum pode também ser considerado o centro arquitetural da MDA. O objetivo disso é permitir que a UML e outros metamodelos MDA reutilizem total ou parcialmente o pacote Núcleo, que permite que outros metamodelos se beneficiem da semântica e da sintaxe abstratas que já foram definidas.

A figura 17.3 ilustra, por meio de um diagrama de pacotes, o relacionamento entre o pacote Núcleo e os demais metamodelos.

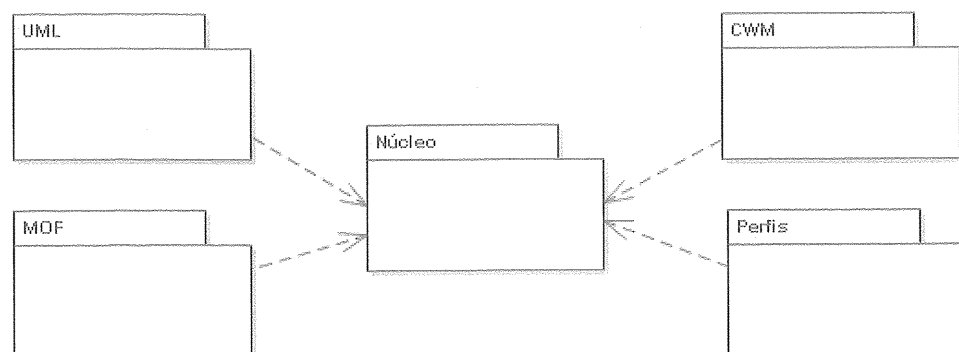


Figura 17.3 – Núcleo UML e Demais Metamodelos.

Como podemos perceber nessa figura, todos os metamodelos apresentam um grau de dependência em relação ao pacote Núcleo.

De maneira a facilitar a reutilização, o pacote Núcleo é subdividido em quatro pacotes, conhecidos como Tipos Primitivos, Abstrações, Básico e Construções, conforme ilustra a figura 17.4.

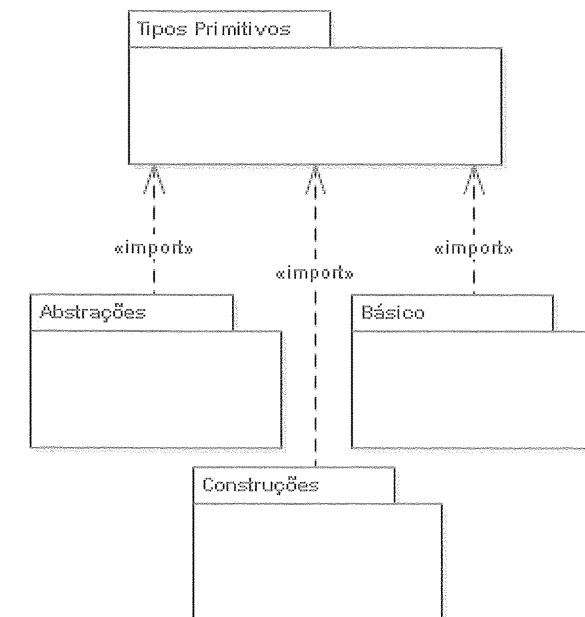


Figura 17.4 – Subdivisões do Pacote Núcleo.

A seguir descreveremos cada um dos pacotes em que se subdivide o pacote Núcleo. Alguns desses pacotes dividem-se, por sua vez, em pacotes menores mais bem-definidos.

- **Tipos Primitivos** – contém um pequeno número de tipos predefinidos que são comumente usados durante a metamodelagem. Esse pacote foi projetado especificamente para satisfazer às necessidades dos metamodelos UML e MOF.
- **Abstrações** – contém a maioria das metaclasses abstratas que podem ser especializadas ou reutilizadas por outros metamodelos. Esse pacote é subdividido em diversos pacotes menores.
- **Construções** – contém a maioria das metaclasses concretas utilizadas principalmente para modelagem orientada a objetos.
- **Básico** – representa as poucas construções usadas como base para produzir XMI para UML, MOF e outros metamodelos baseados na biblioteca de infraestrutura.

Em sua segunda funcionalidade, o pacote Núcleo é usado para definir as construções de modelagem utilizadas para criar metamodelos. Isso é feito por meio da instanciação de metaclasses na Biblioteca de Infraestrutura. Enquanto a instanciação de metaclasses é realizada por meio da MOF, a Biblioteca de Infraestrutura define as metaclasses atuais que são usadas para instanciar os elementos de UML, MOF, CWM e os elementos da própria Biblioteca de Infraestrutura.

17.2.4 Perfis

O pacote Perfis define os mecanismos usados para adaptar metamodelos existentes a plataformas ou domínios específicos. Um perfil deve estar baseado em um metamodelo como a UML, e não é muito útil sozinho. O uso de perfis permite aos usuários criarem “dialetos” da UML, de forma que os mesmos estejam mais adequados às tecnologias de desenvolvimento adotadas por uma determinada empresa ou mesmo à sua área de negócio. Além disso, a utilização de perfis permitirá a adaptação da UML ao surgimento de novas tecnologias, sem a necessidade de se alterar o núcleo da linguagem, o que não ocorria em suas versões anteriores, não sendo estas totalmente adequadas a determinadas tecnologias por não acompanharem suas inovações ou não serem compatíveis com algumas de suas características.

17.2.5 Alinhamento Arquitetural entre a UML e a MOF

Um dos principais objetivos da Infraestrutura da UML tem sido alinhar arquiteturalmente a UML e a MOF. A primeira abordagem utilizada para atingir esse objetivo constituiu-se em definir o núcleo comum, representado pelo pacote Núcleo (Core). Esse pacote permite que os elementos de modelo sejam compartilhados entre a UML e a MOF. A segunda abordagem constituiu-se em definir a UML como um modelo baseado no metamodelo MOF, conforme é apresentado na figura 17.5.

Observe que a MOF não serve de metamodelo somente para a UML, podendo ser utilizada como um metamodelo-base para outros modelos. É importante destacar que todo elemento de modelo da UML é uma instância de um elemento de modelo da MOF.

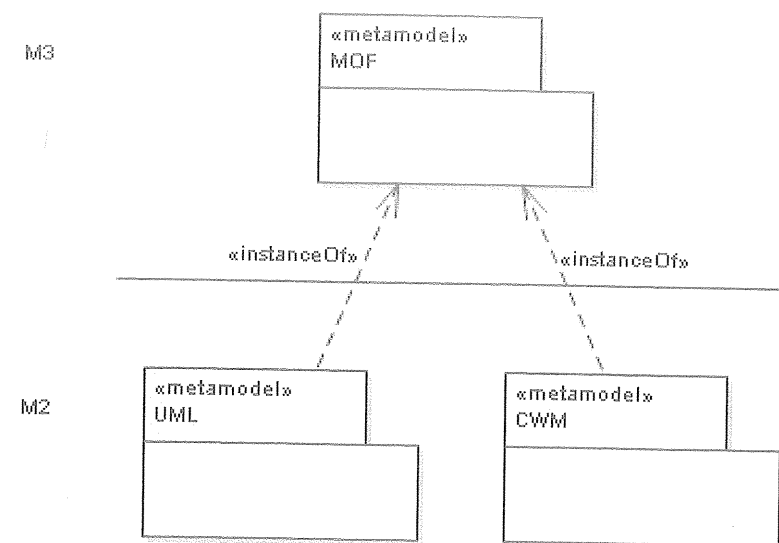


Figura 17.5 – UML e MOF em metaníveis diferentes.

17.2.6 Reutilizando a Infraestrutura

Um dos usos primários da especificação de infraestrutura da UML 2 é que ela deve ser reutilizada quando da criação de outros metamodelos. O metamodelo UML reutiliza a Biblioteca de Infraestrutura de duas maneiras:

- Todo o metamodelo UML é instanciado a partir das meta-metaclasses, que são definidas na Biblioteca de Infraestrutura.
- O metamodelo UML importa e especializa todas as metaclasses da Biblioteca de Infraestrutura.

17.2.7 O Pacote Central (Kernel)

A Biblioteca de Infraestrutura é principalmente reutilizada no pacote de Classes Central (Kernel) na Superestrutura da UML 2. Isso é feito pela união dos diferentes pacotes da infraestrutura, utilizando-se fusão (merge) de pacotes. O pacote Central está exatamente no centro da UML, e as metaclasses de todos os demais pacotes são direta ou indiretamente dependentes dele. O pacote Central é muito semelhante ao pacote Construções da Biblioteca de Infraestrutura, porém, acrescenta mais funcionalidades às construções de modelagem que não precisavam ser incluídas para propósitos de reutilização ou alinhamento com a MOF.

17.2.8 Camadas do Metamodelo

A arquitetura que está centralizada em torno do pacote Núcleo é uma visão complementar da hierarquia do metamodelo de quatro camadas em que o metamodelo UML tem se baseado tradicionalmente. Quando se lida com metacamadas para definir linguagens, há geralmente três camadas que sempre têm sido levadas em conta:

- A especificação da linguagem ou o metamodelo.
- A especificação do usuário ou o modelo.
- Os objetos do modelo.

Essa estrutura pode ser aplicada recursivamente muitas vezes, sendo que um metamodelo em um caso pode ser um modelo em outro, e isso é o que acontece com a UML e a MOF. A UML é uma especificação de linguagem, ou seja, um metamodelo, a partir da qual os usuários podem definir seus próprios modelos. Da perspectiva da MOF, no entanto, a UML é vista como uma especificação de usuário que está baseada na MOF como uma especificação de linguagem.

17.2.9 A Hierarquia de metamodelo de quatro camadas

A camada de meta-metamodelagem forma a base da hierarquia de metamodelagem. A responsabilidade primária dessa camada é definir a linguagem para especificar um metamodelo. Essa camada é muitas vezes referida como M3, e a MOF é um exemplo de um meta-metamodelo. Um meta-metamodelo é tipicamente mais compacto do que um metamodelo descrito por ele, e muitas vezes um meta-metamodelo define diversos metamodelos. É geralmente desejável que metamodelos e meta-metamodelos relacionados compartilhem construções e filosofias de projeto comuns. No entanto, cada camada pode ser vista independentemente das outras camadas e necessita manter sua própria integridade de projeto.

Um metamodelo é uma instância de um meta-metamodelo, o que significa que todo elemento do metamodelo é uma instância de um elemento no meta-metamodelo. A responsabilidade primária da camada de metamodelo é definir uma linguagem para especificar modelos. Essa camada é muitas vezes referida como M2. A UML é um exemplo de metamodelo. Os metamodelos são tipicamente mais elaborados do que os meta-metamodelos que os descrevem, especialmente quando definem semânticas dinâmicas. O metamodelo UML é uma instância do MOF.

Um modelo é uma instância de um metamodelo. A responsabilidade primária da camada de modelo é definir linguagens que descrevam domínios semânticos, ou seja, permitam aos usuários modelar uma ampla variedade de domínios

de problemas diferentes, como software, processos de negócio e requisitos. As coisas que estão sendo modeladas residem fora da hierarquia do metamodelo. Essa camada é muitas vezes referida como M1. Um modelo de usuário é uma instância do metamodelo UML.

Finalmente, a camada M0 contém as instâncias de tempo de execução dos elementos do modelo definidos em um modelo. Assim, a camada M0 representa os objetos propriamente ditos, instanciados a partir do modelo do usuário.

17.2.10 Metamodelagem

A especificação UML é definida usando uma abordagem de metamodelagem que adapta técnicas de especificação formal. Quando se está metamodelando, estabelece-se uma distinção entre metamodelos e modelos. Um modelo tipicamente contém elementos de modelo. Estes são criados pela instanciação de elementos de modelo a partir um metamodelo. O papel típico de um metamodelo é definir a semântica para a forma de modelar elementos dentro de um modelo sendo instanciado.

Um modelo instanciado a partir de um metamodelo pode por sua vez, ser usado como um metamodelo de outro modelo em uma maneira recursiva. Um exemplo simples de metamodelo e modelo é apresentado na figura 17.6.

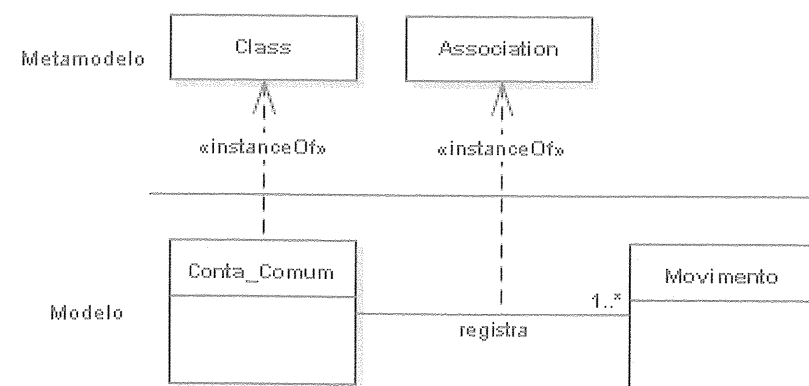


Figura 17.6 – Exemplo Simples de Metamodelo e Modelo.

Nesse exemplo representamos as metaclasses Class e Association que estão definidas no metamodelo UML. São estas metaclasses que permitem a modelagem de classes e associações nos diagramas de classe. Essas metaclasses foram instanciadas em um modelo de usuário de tal forma que as classes Conta_Comum e Movimento são instâncias da metaclasses Class e a associação “registra” entre as classes é uma instância da metaclasses Association. A semântica da UML define

o que acontece quando os elementos de modelo definidos pelo usuário são instanciados no nível M0, gerando nesse caso uma instância de Conta_Corrente e Movimento e uma instância de associação, ou seja, uma ligação entre essas duas instâncias.

17.3 Adaptando e Estendendo a UML

17.3.1 Criação de Perfis

Como foi dito anteriormente, há duas formas de estender a UML, a primeira é por meio da criação de perfis que criam estereótipos adaptando metaclasses originais a domínios específicos. Quando criamos um perfil realizamos uma forma de extensão conservadora, sem alterar muito o metamodelo original, uma vez que adicionamos principalmente estereótipos, restrições e valores rotulados (tags) a ele.

Um perfil permite a adaptação da UML de forma a adequá-la a certos domínios, métodos e/ou tecnologias. Um Perfil é um tipo de Pacote que estende um metamodelo de referência. A construção de extensão primária é o Stereotype (estereótipo), que é definido como parte dos Perfis. Um perfil introduz diversas restrições sobre a metamodelagem ordinária por meio do uso das metaclasses definidas nesse pacote.

Um estereótipo é um tipo limitado de metaclassa que não pode ser usada diretamente, mas deve sempre ser usada em conjunção com uma das metaclasses que ele estende. Cada estereótipo pode estender uma ou mais classes por meio de extensões como parte de um perfil. Similarmente, uma classe pode ser estendida por um ou mais estereótipos.

Dentro do contexto de modelo, um estereótipo permite atribuir novas características a um elemento UML. Já dentro do contexto de metamodelos e perfis, um estereótipo define como uma metaclassa pode ser estendida, atribuindo-lhe novas características e/ou restrições.

No momento em que uma nova metaclassa é derivada a partir de uma metaclassa anterior e utiliza o estereótipo chamado “stereotype”, essa nova metaclassa permite modelar instâncias da metaclassa original contendo um estereótipo com o nome da nova metaclassa. Ou seja, a partir deste momento pode-se modelar elementos estereotipados em um modelo com características que os diferenciem de alguma forma dos elementos originais.

A figura 17.7 apresenta um exemplo simples de como criar perfis. Nessa figura criamos a metaclassa “InternetActor” especializada a partir da metaclassa Actor, aplicando restrições que declarem que um InternetActor deve representar somente atores que acessem o sistema remotamente através da Internet. Observe que aplicamos o estereótipo “stereotype” nesta metaclassa, o que significa que ela deverá ser utilizada associada à metaclassa Actor como um estereótipo.

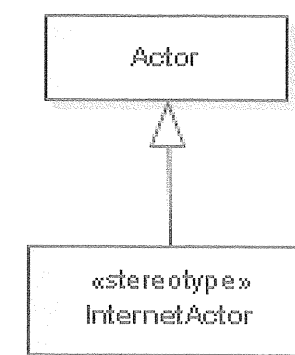


Figura 17.7 – Exemplo Simples de Perfil - Estereótipo InternetActor.

Embora este seja um exemplo bastante simples e, em geral, perfis englobem a criação de vários estereótipos para um determinado domínio, este não deixa de ser um exemplo de perfil, onde a metaclassa Actor foi adaptada para o domínio da Internet. Assim, se aplicássemos esse estereótipo em atores em um diagrama de casos de uso, estes apresentariam o estereótipo de texto “InternetActor”, conforme mostra a figura 17.8.



Figura 17.8 – Ator com o Estereótipo InternetActor.

Esse é um estereótipo de texto, é possível também criar estereótipos gráficos, para isso é preciso associar uma figura ao estereótipo.

17.3.2 Estendendo o Metamodelo

Embora a criação de perfis também estenda a UML, uma vez que novas metaclasses são criadas, elas apenas adaptam um conceito já existente. No exemplo da seção anterior o conceito de ator foi especializado na metaclasses InternetActor, porém apenas adicionamos restrições informando que o ator obrigatoriamente precisa ser um usuário Internet, o conceito de ator permaneceu incólume.

Porém, pode ser necessário criar novas metaclasses para suportarem conceitos ainda não suportados pela UML, sendo possível inclusive criar novas linguagens a partir da linguagem UML se isso for considerado necessário. A criação de novas metaclasses que suportem conceitos novos não é uma extensão conservadora da UML, nesse caso cria-se metaclasses totalmente novas.

Para ilustrar isso, vamos considerar como conceito não suportado pela UML atual o conceito de papel de agente, um conceito utilizado em sistemas multiagentes, ou seja sistemas compostos por agentes de software. Um papel de agente implica em um papel interpretado por um agente que define, entre outras coisas, quais os objetivos, ações e percepções o agente que assumir esse papel deverá possuir.

A metaclasses Actor original não é suficiente para modelar esse conceito, porque está metaclasses representa um tipo de papel executado por uma entidade que interage com o sistema, mas que é externa a ele. No entanto, na maioria das vezes, os agentes de software não são externos ao software, em vez disso, eles costumemente estão inseridos no ambiente do sistema e portanto deveriam ser representados dentro da fronteira do sistema. Dessa forma seria necessário criar uma nova metaclasses que suportasse esse conceito, conforme ilustrado pela figura 17.9, onde criamos a metaclasses AgentRole_Actor para suportar o conceito de papel de agente.

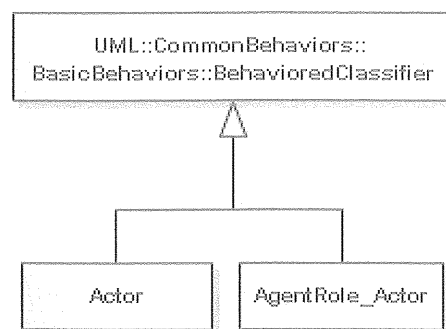


Figura 17.9 – Estendendo o Metamodelo UML por meio da Metaclasses AgentRole_Actor.

Observe que a metaclasses AgentRole_Actor foi derivada a partir da metaclasses BehavoredClassifier, a mesma metaclasses a partir da qual foi derivada a metaclasses

Actor. As metaclasses Actor e AgentRole_Actor possuem muitas semelhanças entre si, porém com algumas diferenças, a metaclasses AgentRole_Actor representa papéis de agente assumidos por entidades internas ao sistema, enquanto a metaclasses Actor representa papéis assumidos por entidades externas ao sistema. Nessa situação não seria possível desenvolver apenas um perfil criando um estereótipo para a metaclasses Actor, porque em um perfil não é possível modificar a semântica das metaclasses originais ou modificar suas restrições, pode-se acrescentar características e restrições novas, mas não se pode retirar ou modificar as originais.

17.4 A Superestrutura da UML 2

O desenvolvimento da superestrutura da UML foi motivado por diversos objetivos, tais como:

- Melhorar o suporte ao desenvolvimento com base em componentes, permitindo tanto a especificação de componentes independentes de plataforma como a especificação de componentes para plataformas específicas.
- Refinar capacidades de especificação arquitetural.
- Melhorar o suporte para desenvolvimento de tempo real.
- Melhorar o suporte para modelagem de processos de negócio.
- Melhorar a escalabilidade e precisão de outras construções comportamentais.
- Aprofundar a precisão de maneira a melhor suportar modelos executáveis.

A superestrutura da UML 2 define as construções no nível de usuário, utilizadas para modelar a estrutura e o comportamento de sistemas. O metamodelo de superestrutura da UML é especificado pelo pacote UML, o qual se divide em diversos pacotes que lidam com modelagem estrutural e comportamental, como pode ser observado na figura 17.10.

Há alguns pacotes que são dependentes entre si, isto ocorre porque as dependências entre os pacotes de nível mais alto apresentam um sumário de todos os relacionamentos entre seus subpacotes.

A superestrutura da UML 2 propõe 13 diagramas para modelagem de sistemas, a maioria dos quais já existia nas versões anteriores ou eram extensões de diagramas já existentes. Os diagramas existentes anteriormente sofreram acréscimos em maior ou menor grau.

Os diagramas da UML 2.0 dividem-se em diagramas estruturais e comportamentais, sendo que esse último tem, ainda, uma subdivisão representada pelos diagramas de interação, conforme pode ser visto na figura 17.11, já apresentada no capítulo 1.

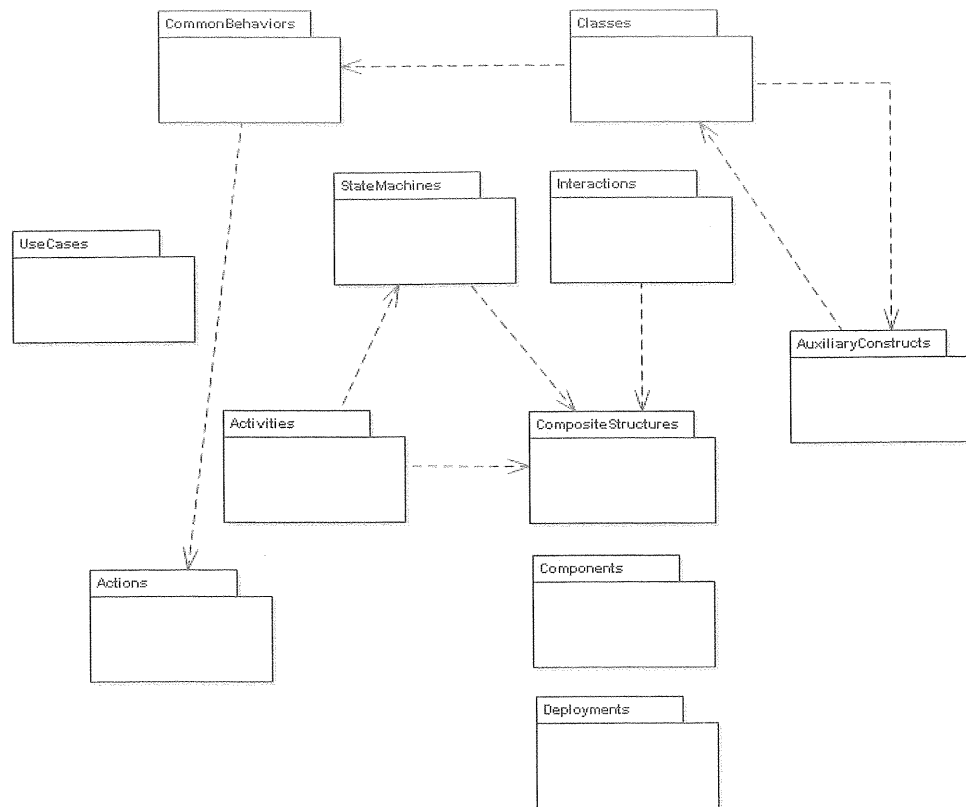


Figura 17.10 – Estrutura de Pacotes da Superestrutura UML 2.

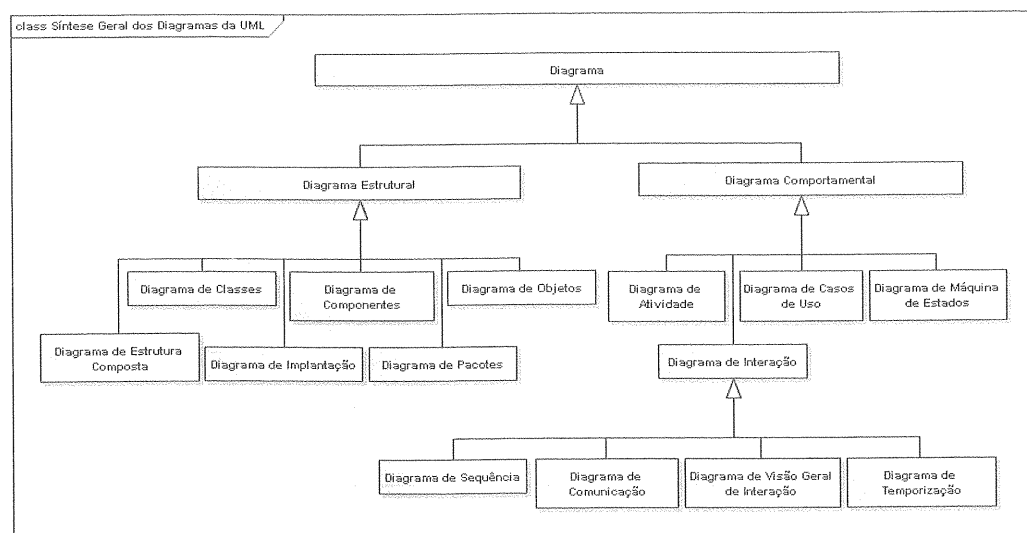


Figura 17.11 – Diagramas da UML.

A

Ação 277, 278, 284, 285, 286, 294, 298, 300
 de chamada de comportamento 289
 de envio de sinal 286, 287, 291
 de estado 247
 de evento de aceitação 286, 287
 de evento de tempo de aceitação 287
 Alfinetes 284, 285
 Análise de requisitos 21, 23, 25
 Âncora 66
 ArgoUML 42
 Arquitetura do sistema 26
 Artefatos 339, 340, 341, 343
 Assinaturas das operações 103, 104, 146
 Associações 57, 58, 106, 337, 377, 378, 380, 381, 385, 386
 binárias 109, 110, 112, 130, 131, 135, 143, 144, 147, 154, 158, 160
 de 1 para 1 157
 de 1 para muitos 158
 de agregação 111, 112, 123, 130, 134
 de composição 112, 113, 143, 144, 150, 154, 178, 350, 351, 380, 381, 384, 386, 395
 de especialização 58, 60, 113, 162, 185
 de extensão 58, 63, 64, 65, 66, 70, 71, 81, 86, 92, 96, 99, 206, 210, 217, 318, 361, 362, 363
 de generalização 58, 60, 113, 162
 de inclusão 58, 61, 71, 72, 99, 206, 210, 218, 219, 318, 361, 362, 364
 de muitos para muitos 159, 161, 167
 qualificadas 124
 ternárias 110, 111, 161
 unárias 107, 157, 158
 Ativação 195
 Atividade 277, 278, 280, 281, 283, 284, 285, 287, 289, 291, 292, 298, 299, 300, 301, 304, 305, 306, 307, 308, 309, 310, 311
 de estado 247, 248
 estruturada 293
 interna 246, 247
 Ator 30, 53, 55, 56, 57, 59, 60, 68, 69, 72, 78, 79, 81, 82, 85, 86, 87, 90, 91, 92, 93, 95, 97, 98
 especializado 60
 geral 60
 principal 56
 secundário 54, 56

Atributo 44, 45, 46, 47, 48, 49, 51, 101, 103, 105, 107, 109, 113, 114, 115, 120, 121, 124, 127, 133, 134, 135, 139, 140, 142, 143, 144, 145, 146, 148, 150, 151, 152, 153, 154, 155, 156, 157, 160, 162, 163, 164, 166, 167, 171, 172, 173, 175, 177, 178, 180, 181, 182, 183, 184
 derivado 105
 estático 121, 148
 protegido 133
 Autochamadas 196, 200, 225, 233
 Autodelegações 200
 Autotransições 248, 249, 275

B

Barra de bifurcação/união 251, 252, 255

C

Camada de persistência 155
 Casos de uso 31, 53, 54, 55, 56, 58, 59, 61, 62, 63, 64, 65, 66, 68, 69, 70, 71, 72, 79, 81, 82, 83, 85, 86, 87, 90, 91, 92, 93, 94, 96, 98, 99
 especializados 56, 59
 gerais 56, 58, 70
 primários 54, 79
 secundários 54, 79, 90, 94, 96, 363
 Chaves
 estrangeiras 109, 157, 158, 160, 161, 164, 166, 167
 primárias 109, 156, 157, 158, 159, 160, 161, 163, 166, 167
 Classes 43, 44, 45, 47, 48, 49, 50, 51, 101, 102, 103, 104, 105, 106, 107, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 122, 124, 125, 126, 127, 128, 130, 131, 133, 134, 135, 137, 138, 139, 140, 142, 143, 144, 145, 146, 147, 148, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 164, 165, 167, 170, 171, 173, 175, 177, 178, 180, 181, 182, 183, 184, 185, 186
 abstratas 45, 115, 137, 152, 162, 163
 associativas 115, 116, 159, 161
 de controle 141, 142, 149, 152, 155, 171, 173, 175, 177, 180, 195, 203, 204, 216, 220, 223, 224, 225, 226, 228, 232, 235, 236, 376, 392, 394, 395, 396, 397, 398, 399

de entidade 126, 127, 128, 137, 141, 142, 145, 149, 152, 171, 173, 175, 177, 180, 215, 216, 223, 224, 225, 226, 228, 232, 377
 de fronteira 128, 141, 171, 173, 175, 177, 180, 195, 216, 220, 223, 224, 225, 226, 228
 de interface 149, 152
 especializadas 113, 114, 124, 125, 152, 162
 filha 48, 113, 163
 gerais 48, 49, 50, 113, 114, 115, 124, 162
 intermediárias 116, 142, 159, 161, 172, 180, 182, 382, 384, 386
 internas 323, 324
 mãe 48, 113
 persistentes 109, 126, 155, 156
 transientes 126, 155

Cláusulas

Do 247, 248, 252, 254, 255
 Entry 246, 247, 252, 254
 Exit 246, 247, 252, 254

Colaboração 40, 345, 347, 348

Coleção 124

ordenada 123

Componentes 38, 320, 321, 322, 323, 324, 325, 326, 329, 330, 331, 332, 341, 343, 344

físicos 320

internos 323, 324

lógicos 320

Comportamentos 46

Condições de guarda 207, 220, 225, 228, 236, 237, 238, 248, 249, 250, 256, 258, 260, 279, 280, 281

Conector de aninhamento 188

Conectores 288, 289, 346, 347, 349, 350

Custos 25

D

Dependências 116, 117, 120, 185, 187, 189, 190, 191, 322, 323, 324, 342, 348, 388

Detalhes de tempo 200, 201

Diagrama

de atividade 36, 277, 281, 282, 286, 294, 296, 297, 298, 300, 311

de casos de uso 30, 52, 53, 69, 72, 78, 79, 82, 84, 85, 90, 92, 94, 96, 98, 359, 360, 362

de classes 31, 32, 101, 102, 103, 106, 131, 132, 136, 138, 141, 145, 149, 151, 152, 155, 171, 173, 175, 177, 179, 375

de componentes 38, 320, 324, 325, 328, 330, 331, 332

de comunicação 35, 231, 235, 236

de estrutura composta 40, 345

de implantação 39, 334, 337, 341, 342

de máquina de estados 35, 242, 252, 260, 263, 264, 265, 266

de objetos 32, 183, 184, 185, 186, 387

de pacotes 33, 187, 388

de sequência 33, 192, 193, 194, 197, 202, 203, 206, 207, 215, 216, 218, 219, 231, 313, 317, 389, 391, 393, 394, 395, 396, 397, 398, 399, 400, 401

de tempo 40, 352, 353

de visão geral de interação 37, 313, 314, 315, 318

Diagramas

comportamentais 41

de interação 41

estruturais 41

Direção de leitura 110

Documentação 21, 26, 28, 29

de casos de uso 55

histórica 29

E

Edição para a comunidade 41, 42

Enterprise Architect 41

Especificação de Implantação 340

Estado 242, 243, 244, 245, 246, 247, 249, 250, 252, 254, 255, 256, 257, 258, 259, 261, 262, 263, 264, 265, 270, 271, 273, 275, 353

composto 255, 256, 257, 259, 260, 261,

262, 267, 270, 273, 275

composto ortogonal 257

de história 257

de sincronismo 258, 259

de submáquina 259, 260, 261, 267, 275, 276

estático 243, 258

final 244, 255

inicial 244, 256

interno 243

invariante 213

ortogonal 255, 257

simples 243, 267

Estereótipo 61, 63, 68, 116, 117, 123, 126, 127, 129, 130, 131, 155, 191, 232, 321, 322, 336, 337, 338

<<asp page>> 131

<<boundary>> 126, 128, 136, 376

<<build>> 131

<<client page>> 129, 130

<<column>> 157

<<computer>> 336

<<control>> 126, 128, 137, 376

<<datastore>> 288

de controle 231

de fronteira 128, 231

<<delegate>> 324

de texto 68

<<device>> 334, 336

<<document>> 321

<<entity>> 126

<<enumeration>> 131

<<executable>> 321

<<ExecutionEnvironment>> 334

<<file>> 321

<<form>> 129, 130, 131

<<framework>> 191

gráfico 68, 127, 129, 156, 191, 321, 336

<<import>> 189

<<instantiate>> 185

<<jsp page>> 131

<<library>> 321

<<link>> 130

<<manifest>> 339

<<merge>> 189

<<model>> 191

<<occurrence>> 348

<<persistente>> 127

<<PK>> 157, 158

<<represents>> 348

<<secure>> 336

<<server>> 336

<<server page>> 129

<<servlet>> 131

<<storage>> 336

<<structure>> 292

<<submit>> 131

<<subsystem>> 191

<<system>> 191

<<table>> 156, 321

<<unique>> 158

<<web page>> 131

Estímulos 196

Etiquetas 335

Exceções 286

F

Ferramentas CASE 41

Final de fluxo 283

Fluxo

alternativo 57, 66, 208

principal 57

Fluxos

de controle 277, 278, 279, 283, 284, 286

de dados 278

de exceção 57, 291

de interrupção 286, 291

de objetos 277, 279, 284, 285, 286, 288, 296, 297, 300, 301, 304, 305, 307, 308, 309, 311, 312

paralelos 208

Foco de controle 195, 198, 231

Fragmentos

combinados 207, 208, 209, 210, 211, 212, 213, 217, 220, 223, 225, 227, 229, 230, 392, 394, 395, 396, 397, 398, 399, 400

de interação 203, 204, 206, 214

Frameworks de persistência 156

Fronteira do sistema 69

Funcionalidades 22, 23, 27, 31, 52, 53, 54, 56, 58, 64, 69, 315, 359, 360, 361, 362, 394, 396

H

Herança 48, 113, 162, 163, 164

múltipla 50

I

Instanciação 44

Interfaces 118, 202

fornecidas 118, 120, 322

requeridas 119, 120, 322, 329, 331

Iterações 236, 237, 238

L

Levantamento de requisitos 21, 22, 23

Lifelines 193, 194, 195, 231, 232

Linha

de vida de estado 352, 353

de vida de valor 352

Linhas de vida 193, 194, 195, 197, 198, 200, 202, 205, 210, 214, 218, 219, 231, 352, 395

M

Manifestação 340, 341, 344

Manutenção 27, 28

Mapeamento de classes 155

Máquina

de estado comportamental 35, 242

de estado de protocolo 35, 242

de subestado 255, 267

Mensagens 196, 197, 198, 199, 200, 205, 206, 213, 218, 233, 235, 236, 237

de retorno 197, 199, 200, 237

externas 202

internas 202

Métodos 45, 46, 47, 48, 49, 50, 51, 101, 102, 103, 105, 109, 113, 114, 116, 120, 127, 132, 136, 137, 138, 139, 140, 141, 142, 143, 144, 146, 148, 149, 150, 151, 153, 154, 155, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 192, 377, 378, 380, 381, 383, 384, 385, 387, 389

construtores 137, 138, 198

destrutores 138, 198, 199, 201, 395

polimórficos 51

Modelo

conceitual 101, 131, 132, 159, 165, 166, 167, 192

de domínio 101, 132, 136, 141, 145, 149, 151, 152, 170, 172, 174, 176, 179, 192, 375, 387

Multiplicidade 68, 105, 107, 108, 109, 115, 134, 135, 142, 152, 154, 157, 159, 160, 166, 167, 350, 383, 384

N

Navegabilidade 109, 131, 132, 136

Nó 285, 334, 335, 336, 337, 338, 340, 342, 343

com denominação 335

com detalhes de configuração 335

de ação 278, 279, 281, 283, 284, 285, 291, 296, 298, 301, 304, 306

de atividade estruturada 292

de atividade executável 292

de bifurcação/união 251, 281, 282, 283, 290, 291, 300, 307, 314, 316, 319

de buffer central 288

de controle 280, 282

de decisão 280, 281, 294, 299, 300, 304, 305, 306, 307, 309, 310, 311, 312, 313, 314, 316, 317, 319, 455

de entrada de exceção 286

de expansão 293

de final de atividade 280

de objeto 284, 285, 286, 288, 293

de parâmetro de atividade 285, 298, 299, 300

de parâmetro de entrada 285

de parâmetro de saída 285

de repositório de dados 288

inicial 280, 313

Nota explicativa 65, 195, 244, 260

O

Objetos 43, 44, 45, 183, 184, 185, 193, 195, 196, 197, 198, 200, 201, 202, 203, 205, 210, 215, 218, 219, 220, 223, 224, 225, 226, 228, 231, 232, 233, 234, 235, 278, 284, 286, 291, 388

Objetos-parte 111, 112, 143, 144, 380, 381, 384, 386

Objetos-todo 111, 112, 143, 144, 380, 381, 384, 386

OCL 120

Ocorrências

de colaboração 348, 349

de interação 204, 205, 206, 207, 210, 217, 218, 230, 392, 395, 397, 399

OMG 20

Operações 46, 101

Operador

de interação 207, 208, 210, 211, 212

de interação Alt 207, 208, 395

de interação Assertion 213

de interação Break 211

de interação Consider 213

de interação Critical Region 212, 214

de interação Ignore 213

de interação Loop 211, 220, 223, 225, 226, 227, 229, 230, 394, 395, 396, 397, 398, 399, 400

de interação Neg 212

de interação Opt 208, 209, 210, 217, 228, 230, 395, 397, 398, 399

de interação Par 210

de interação Seq 213

de interação Strict 213

de interação Weak Sequencing 213
Ref 204

Operando de interação 208, 210

Orientação a objetos 43

P

Pacotes 187, 189, 190, 191, 342

Papéis 108, 345, 346, 347, 348

Partes 350

internas 349

Partições de atividade 290

Persistência 155

Peso 279

Pins 284

Polimorfismo 50, 51, 139

Ponto

de extensão 66

final 202

Portão 206, 207

Portas 117, 202, 324, 349, 350

Poseidon for UML 42

Prazos 25

Processo Unificado 22

Projeto 21, 26

navegacional 129, 130

Propriedade referenciada 351

Propriedades 45, 350

Prototipação 24, 25

Pseudoestado

de escolha 249, 256, 267, 271, 273, 275, 276

de história 256, 257

de história profunda 257

de junção 260, 261, 267, 273, 275, 276

de ponto de entrada 261, 262

de ponto de saída 261, 262

de término 263

inicial 255, 256

Q

Quadro de interação 313, 314, 317

Quadros de ocorrência de interação 313, 314, 315, 317

Qualificador 124

R

RAD 24

Realização 117, 120

Região 255, 257, 258, 293

de atividade 291

de expansão 293

ortogonal 255

Regras de negócio 22, 57, 66

Requisitos

funcionais 22, 54

não-funcionais 22

Restrição 65, 120, 121, 122, 124, 126, 147, 152, 155, 213, 279

bag 123

completa 124, 125

de duração 200, 353

de interação 207, 209, 210, 211

de tempo de execução 213

disjunta 124

em atributos 121

incompleta 124, 125

ordered 123

readOnly 121, 133

separada 124, 125

Seq 123

sobreposta 125

unique 123

Reusabilidade 29

S

Separador de operando de interação 208

Sinais de controle 279

Sinal 279

Subclasses 48, 49, 50, 51, 113, 124, 125, 133, 134, 135, 155

Subestados 243, 255, 256, 257, 259, 260, 270, 273, 275

internos 259

Submáquina de estado 264

Superclasses 48, 49, 50, 51, 113, 135, 164
abstrata 134

T

Tabelas 155, 156, 157, 158, 160, 161, 162, 163, 164, 165, 166, 321

intermediárias 159, 160, 161, 167

Tags 335

Transições 243, 244, 245, 247, 248, 249, 252,
254, 255, 256, 257, 258, 259, 260, 262,
264, 265, 267, 270, 271, 273, 275, 276,
353

concorrentes 251

de estado 242

internas 248, 256

U

Uso de interação 204, 392

V

Valores iniciais 105

Vínculos 184, 185, 186, 232, 234, 235

Visão 183, 184, 185

de caixa branca 323, 324

de caixa preta 323

Visibilidade 47

pacote 47

privada 47, 108

protegida 47, 108

pública 47, 108

Visual Paradigm for UML 42

VP-UML 42